

## Introducción a los sistemas operativos

Si estás tomando un curso universitario de sistemas operativos, ya deberías tener una idea de lo que hace un programa de computadora cuando se ejecuta.

De lo contrario, este libro (y el curso correspondiente) va a ser difícil, por lo que probablemente deberías dejar de leerlo o correr a la librería más cercana y consumir rápidamente el material básico necesario antes de continuar (tanto Patt como Patel [PP03] y Bryant & O'Hallaron [BOH10] son libros bastante buenos).

Entonces, ¿qué sucede cuando se ejecuta un programa?

Bueno, un programa en ejecución hace una cosa muy simple: ejecuta instrucciones. Muchos millones (y hoy en día, incluso miles de millones) de veces por segundo, el procesador recupera una instrucción de la memoria, la decodifica (es decir, descubre qué instrucción es) y la ejecuta (es decir, hace lo que desea). Se supone que debe hacer, como sumar dos números, acceder a la memoria, verificar una condición, saltar a una función, etc.). Una vez terminada esta instrucción, el procesador pasa a la siguiente instrucción, y así sucesivamente, hasta que el programa finalmente se completa<sup>1</sup>.

Así, acabamos de describir los fundamentos del modelo de computación de Von Neumann<sup>2</sup>. Suena simple, ¿verdad? Pero en esta clase aprenderemos que mientras se ejecuta un programa, suceden muchas otras cosas locas con el objetivo principal de hacer que el sistema sea fácil de usar.

De hecho, existe una gran cantidad de software que es responsable de facilitar la ejecución de programas (incluso permitiéndole aparentemente ejecutar muchos al mismo tiempo), permitiendo que los programas compartan memoria, permitiendo que los programas interactúen con dispositivos y otras funciones divertidas. cosas así. Ese cuerpo de software

---

<sup>1</sup>Por supuesto, los procesadores modernos hacen muchas cosas extrañas y aterradoras bajo el capó para hacer que los programas se ejecuten más rápido, por ejemplo, ejecutar múltiples instrucciones a la vez, ¡e incluso emitirlas y completarlas sin orden! Pero esa no es nuestra preocupación aquí; sólo nos preocupa el modelo simple que asumen la mayoría de los programas: que las instrucciones aparentemente se ejecutan una a la vez, de manera ordenada y secuencial.

<sup>2</sup>Von Neumann fue uno de los primeros pioneros de los sistemas informáticos. También realizó un trabajo pionero en teoría de juegos y bombas atómicas, y jugó en la NBA durante seis años. Bien, una de esas cosas no es cierta.

**EL FUNDAMENTO DEL PROBLEMA:  
CÓMO VIRTUALIZAR LOS RECURSOS**

Una pregunta central que responderemos en este libro es bastante simple: ¿cómo virtualiza el sistema operativo los recursos? Éste es el meollo de nuestro problema. Por qué el sistema operativo hace esto no es la pregunta principal, ya que la respuesta debería ser obvia: hace que el sistema sea más fácil de usar. Por lo tanto, nos centramos en el cómo: ¿qué mecanismos y políticas implementa el sistema operativo para lograr la virtualización? ¿Cómo lo hace el sistema operativo de manera tan eficiente? ¿Qué soporte de hardware se necesita?

Usaremos el "crud del problema", en cuadros sombreados como este, como una forma de señalar problemas específicos que estamos tratando de resolver al construir un sistema operativo. Por lo tanto, dentro de una nota sobre un tema en particular, puede encontrar una o más cruces (sí, este es el plural adecuado) que resaltan el problema. Los detalles del capítulo, por supuesto, presentan la solución, o al menos los parámetros básicos de una solución.

Se denomina sistema operativo (SO), el <sup>3</sup>, ya que es el encargado de asegurarse de que sistema funciona de manera correcta y eficiente de una manera fácil de usar.

La forma principal en que el sistema operativo hace esto es a través de una técnica general que llamamos virtualización. Es decir, el sistema operativo toma un recurso físico (como el procesador, la memoria u un disco) y lo transforma en una forma virtual de sí mismo más general, potente y fácil de usar . Por eso, a veces nos referimos al sistema operativo como una máquina virtual.

Por supuesto, para permitir a los usuarios decirle al sistema operativo qué hacer y así hacer uso de las funciones de la máquina virtual (como ejecutar un programa, asignar memoria o acceder a un archivo), el sistema operativo también proporciona algunas interfaces (API) a las que puede llamar. De hecho, un sistema operativo típico exporta unos cientos de llamadas al sistema que están disponibles para las aplicaciones. Debido a que el sistema operativo proporciona estas llamadas para ejecutar programas, acceder a la memoria y a los dispositivos, y otras acciones relacionadas, a veces también decimos que el sistema operativo proporciona una biblioteca estándar para las aplicaciones.

Finalmente, debido a que la virtualización permite que muchos programas se ejecuten (compartiendo así la CPU), y muchos programas accedan simultáneamente a sus propias instrucciones y datos (compartiendo así memoria), y muchos programas accedan a dispositivos (compartiendo así discos, etc.). ), el sistema operativo a veces se conoce como administrador de recursos. Cada uno de la CPU, la memoria y el disco es un recurso del sistema; Por lo tanto, la función del sistema operativo es administrar esos recursos, haciéndolo de manera eficiente o justa o incluso con muchos otros objetivos posibles en mente. Para comprender un poco mejor el papel del sistema operativo, veamos algunos ejemplos.

---

<sup>3</sup>Otro de los primeros nombres del sistema operativo fue supervisor o incluso programa de control maestro. Aparentemente, esto último sonó un poco demasiado entusiasta (ver la película Tron para más detalles) y por eso, afortunadamente, el "sistema operativo" se hizo popular.

```

1 #incluir <stdio.h>
2 #incluir <stdlib.h>
3 #incluir <sys/time.h>
4 #incluir <afirmar.h>
5 #incluir "común.h"
6
7 enteros
8 principal(int argc, char *argv[])
9 {
10     si (argc != 2) {
11         fprintf(stderr, "uso: cpu <cadena>\n");
12         salir(1);
13     }
14     char *cadena = argv[1];
15     mientras (1) {
16         Girar(1);
17         printf("%s\n", cadena);
18     }
19     devolver 0;
20 }
```

Figura 2.1: Ejemplo simple: código que se repite e imprime (cpu.c)

## 2.1 Virtualizando la CPU

La Figura 2.1 muestra nuestro primer programa. No hace mucho. De hecho, todos lo que hace es llamar a `Spin()`, una función que verifica repetidamente la hora y regresa una vez que se ha ejecutado por un segundo. Luego, imprime la cadena que el El usuario pasó en la línea de comando y se repite para siempre.

Digamos que guardamos este archivo como `cpu.c` y decidimos compilarlo y ejecutarlo. en un sistema con un solo procesador (o CPU como a veces lo llamaremos). Esto es lo que veremos:

```

símbolo> gcc -o cpu cpu.c -Wall
símbolo> ./cpu "A"
A
A
A
A
DO
mensaje>
```

Una ejecución no demasiado interesante: el sistema comienza a ejecutar el programa, que comprueba repetidamente el tiempo hasta que transcurre un segundo. Una vez que ha pasado un segundo, el código imprime la cadena de entrada pasada por el usuario (en este ejemplo, la letra "A"), y continúa. Tenga en cuenta que el programa se ejecutará para siempre; Al presionar "Control-c" (que en sistemas basados en UNIX finalizará el programa que se ejecuta en primer plano) podemos detener el programa.

```
símbolo> ./cpu A y ./cpu B y ./cpu C y ./cpu D y [1] 7353 [2] 7354 [3] 7355 [4] 7356 A
```

```
B  
D  
do  
A  
B  
D  
do  
A  
...  
...
```

Figura 2.2: Ejecutando muchos programas a la vez

Ahora hagamos lo mismo, pero esta vez ejecutemos muchas instancias diferentes de este mismo programa. La figura 2.2 muestra los resultados de este ejemplo un poco más complicado.

Bueno, ahora las cosas se están poniendo un poco más interesantes. Aunque sólo tenemos un procesador, ¡de alguna manera los cuatro programas parecen estar ejecutándose al mismo tiempo! ¿Cómo ocurre esta magia?<sup>4</sup>

Resulta que el sistema operativo, con algo de ayuda del hardware, está a cargo de esta ilusión, es decir, la ilusión de que el sistema tiene una gran cantidad de CPU virtuales. Convertir una única CPU (o un pequeño conjunto de ellas) en un número aparentemente infinito de CPU y permitir así que muchos programas se ejecuten aparentemente a la vez es lo que llamamos virtualizar la CPU, el enfoque de la primera parte importante de este libro.

Por supuesto, para ejecutar programas, detenerlos y decirle al sistema operativo qué programas ejecutar, es necesario que haya algunas interfaces (API) que pueda usar para comunicar sus deseos al sistema operativo. Hablaremos de estas API a lo largo de este libro; de hecho, son la forma principal en la que la mayoría de los usuarios interactúan con los sistemas operativos.

También puede notar que la capacidad de ejecutar múltiples programas a la vez plantea todo tipo de preguntas nuevas. Por ejemplo, si dos programas quieren ejecutarse en un momento determinado, ¿cuál debería ejecutarse? Esta pregunta es respondida por una política del sistema operativo; Las políticas se utilizan en muchos lugares diferentes dentro de un sistema operativo para responder este tipo de preguntas y, por lo tanto, las estudiaremos a medida que aprendamos sobre los mecanismos básicos que implementan los sistemas operativos (como la capacidad de ejecutar múltiples programas a la vez). De ahí el papel del sistema operativo como administrador de recursos.

---

<sup>4</sup>Observe cómo ejecutamos cuatro procesos al mismo tiempo, usando el símbolo &. Al hacerlo, se ejecuta un trabajo en segundo plano en el shell zsh , lo que significa que el usuario puede emitir inmediatamente su siguiente comando, que en este caso es otro programa para ejecutar. Si está utilizando un shell diferente (por ejemplo, tcsh), funciona de manera ligeramente diferente; lea la documentación en línea para obtener más detalles.

```

1 #incluir <unistd.h>
2 #incluir <stdio.h>
3 #incluir <stdlib.h>
4 #incluir "común.h"
5
6 enteros
7 principal(int argc, char *argv[])
8 {
9     int *p = malloc(tamaño de(int)); afirmar(p!= NULL); // a1
10
11    printf("(%) dirección señalada por p: %p\n",
12          getpid(), p); *p = 0; // a2
13    mientras
14    (1) {
15        Girar(1);
16        *p = *p + 1;
17        printf("(%) p: %d\n", getpid(), *p); // a3
18    }
19    devolver 0;
20 }

```

Figura 2.3: Un programa que accede a la memoria (mem.c)

## 2.2 Virtualización de la memoria

Ahora consideremos la memoria. El modelo de memoria física que presentan las máquinas modernas es muy sencillo. La memoria es sólo una serie de bytes; Para leer la memoria, se debe especificar una dirección para poder acceder. los datos allí almacenados; Para escribir (o actualizar) la memoria, también se debe especificar los datos se escribirán en la dirección indicada.

Se accede a la memoria todo el tiempo cuando se ejecuta un programa. Un programa mantiene todas sus estructuras de datos en la memoria y accede a ellas a través de varias instrucciones, como cargas y almacenes u otras instrucciones explícitas que acceden a la memoria al realizar su trabajo. No olvides que cada instrucción del programa también está en la memoria; por lo tanto se accede a la memoria en cada búsqueda de instrucciones.

Echemos un vistazo a un programa (en la Figura 2.3) que asigna algo de memoria llamando a malloc(). El resultado de este programa se puede encontrar aquí:

```

mensaje> ./mem
(2134) dirección señalada por p: 0x200000
(2134) pági: 1
(2134) pági: 2
(2134) pági: 3
(2134) pági: 4
(2134) pági: 5
DO

```

```

mensaje> ./mem &; ./mem & [1] 24113
[2] 24114
(24113)
dirección apuntada por p: 0x200000 (24114) dirección apuntada por
p: 0x200000 (24113) p: 1 (24114) p: 1 (24114) p: 2 (24113) p: 2
(24113) p: 3 (24114)
p: 3 (24113) p: 4
(24114) p: 4
...

```

Figura 2.4: Ejecutar el programa de memoria varias veces

El programa hace un par de cosas. Primero, asigna algo de memoria (línea a1). Luego, imprime la dirección de la memoria (a2) y luego coloca el número cero en la primera ranura de la memoria recién asignada (a3). Finalmente, realiza un bucle, se retrasa un segundo e incrementa el valor almacenado en la dirección contenida en p. Con cada declaración de impresión, también imprime lo que se llama el identificador de proceso (PID) del programa en ejecución.

Este PID es único por proceso en ejecución.

De nuevo, este primer resultado no es demasiado interesante. La memoria recién asignada está en la dirección 0x200000. A medida que se ejecuta el programa, actualiza lentamente el valor e imprime el resultado.

Ahora, volvemos a ejecutar varias instancias de este mismo programa para ver qué sucede (Figura 2.4). Vemos en el ejemplo que cada programa en ejecución ha asignado memoria en la misma dirección (0x200000) y, sin embargo, cada uno parece estar actualizando el valor en 0x200000 de forma independiente. Es como si cada programa en ejecución tuviera su propia memoria privada, en lugar de compartir la misma memoria física con otros programas en ejecución<sup>5</sup>.

De hecho, eso es exactamente lo que está sucediendo aquí mientras el sistema operativo está virtualizando la memoria. Cada proceso accede a su propio espacio de direcciones virtuales privado (a veces llamado simplemente espacio de direcciones), que el sistema operativo de alguna manera asigna a la memoria física de la máquina. Una referencia a la memoria dentro de un programa en ejecución no afecta el espacio de direcciones de otros procesos (o el propio sistema operativo); En lo que respecta al programa en ejecución, tiene memoria física para sí mismo. La realidad, sin embargo, es que la memoria física es un recurso compartido, gestionado por el sistema operativo. Cómo se logra exactamente todo esto es también el tema de la primera parte de este libro, sobre el tema de la virtualización.

---

<sup>5</sup>Para que este ejemplo funcione, debe asegurarse de que la aleatorización del espacio de direcciones esté deshabilitada; Resulta que la aleatorización puede ser una buena defensa contra ciertos tipos de fallas de seguridad. Lea más sobre esto por su cuenta, especialmente si desea aprender cómo ingresar a sistemas informáticos mediante ataques de destrucción de pilas. No es que recomendáramos tal cosa...

## 2.3 Concurrency

```

1 #incluir <stdio.h>
2 #incluir <stdlib.h>
3 #incluir "común.h"
4 #incluir "common_threads.h"

5
6 contador int volátil = 0;
7 bucles internos;
8
9 vacío *trabajador(vacio *arg) {
10     ent yo;
11     for (i = 0; i < bucles; i++) {
12         contador++;
13     }
14     devolver NULO;
15 }
16
17 int principal(int argc, char *argv[]) {
18     si (argc != 2) {
19         fprintf(stderr, "uso: subprocessos <valor>\n");
20         salir(1);
21     }
22     bucles = atoi(argv[1]);
23     pthead_t p1, p2;
24     printf("Valor inicial: %d\n", contador);
25
26     Pthread_create(&p1, NULL, trabajador, NULL);
27     Pthread_create(&p2, NULL, trabajador, NULL);
28     Pthread_join(p1, NULO);
29     Pthread_join(p2, NULO);
30     printf("Valor final: %d\n", contador);
31     devolver 0;
32 }
```

Figura 2.5: Un programa multiproceso (threads.c)

Otro tema principal de este libro es la concurrencia. Usamos este término conceptual para referirnos a una serie de problemas que surgen y deben abordarse, cuando se trabaja en muchas cosas a la vez (es decir, simultáneamente) en el mismo programa. Los problemas de concurrencia surgieron primero dentro del sistema operativo. sistema mismo; Como puede ver en los ejemplos anteriores sobre virtualización, el El sistema operativo hace malabarismos con muchas cosas a la vez: primero ejecuta un proceso, luego otro, y así sucesivamente. Resulta que hacerlo conduce a algunos problemas profundos y problemas interesantes.

Desafortunadamente, los problemas de concurrencia ya no se limitan sólo al propio sistema operativo. De hecho, los programas multiproceso modernos exhiben la mismos problemas. Demostremos con un ejemplo de subprocesos múltiples programa (Figura 2.5).

Aunque es posible que no entiendas completamente este ejemplo en este momento (y aprenderemos mucho más sobre él en capítulos posteriores, en la sección del libro sobre concurrencia), la idea básica es simple. El programa principal crea dos subprocesos usando `Pthread create()`<sup>6</sup>. Puedes pensar en un hilo como una función que se ejecuta dentro del mismo espacio de memoria que otras funciones, con más de una activa a la vez. En este ejemplo, cada hilo comienza a ejecutarse en una rutina llamada `trabajador()`, en la que simplemente incrementa un contador en un bucle `for loops` varias veces.

A continuación se muestra una transcripción de lo que sucede cuando ejecutamos este programa con el valor de entrada para los bucles variables establecido en 1000. El valor de los bucles determina cuántas veces cada uno de los dos trabajadores incrementará el contador compartido en un bucle. Cuando el programa se ejecuta con el valor de los bucles establecido en 1000, ¿cuál espera que sea el valor final del contador ?

```
indicador> gcc -o hilo thread.c -Wall -pthread indicador> ./thread 1000 Valor inicial: 0 Valor
final: 2000
```

Como probablemente habrás adivinado, cuando los dos subprocesos finalizan, el valor final del contador es 2000, ya que cada subproceso incrementó el contador 1000 veces. De hecho, cuando el valor de entrada de los bucles se establece en N, esperaríamos que la salida final del programa fuera  $2N$ . Pero resulta que la vida no es tan simple. Ejecutemos el mismo programa, pero con valores más altos para los bucles, y veamos qué sucede:

```
indicador> ./thread 100000 Valor inicial:
0 Valor final: 143012
indicador> ./thread 100000 // ¿¿eh??
Valor inicial: 0
Valor final: 137298 // ¿Qué diablos?
```

En esta ejecución, cuando dimos un valor de entrada de 100 000, en lugar de obtener un valor final de 200 000, primero obtenemos 143 012. Luego, cuando ejecutamos el programa por segunda vez, no solo obtenemos nuevamente el valor incorrecto, sino también un valor diferente al de la última vez. De hecho, si ejecuta el programa una y otra vez con valores altos de bucles, es posible que a veces incluso obtenga la respuesta correcta. Entonces, ¿por qué sucede esto?

Resulta que la razón de estos resultados extraños e inusuales se relaciona con cómo se ejecutan las instrucciones, que es una a la vez. Desafortunadamente, una parte clave del programa anterior, donde se incrementa el contador compartido,

---

<sup>6</sup>La llamada real debe ser `pthread create()` en minúsculas ; la versión en mayúsculas es nuestro propio contenedor que llama a `pthread create()` y se asegura de que el código de retorno indique que la llamada se realizó correctamente. Consulte el código para obtener más detalles.

**EL CRUZ DEL PROBLEMA: CÓMO  
CONSTRUIR PROGRAMAS CONCURRENTES CORRECTOS**

Cuando hay muchos subprocesos ejecutándose simultáneamente dentro del mismo espacio de memoria, ¿cómo podemos construir un programa que funcione correctamente? ¿Qué primitivas se necesitan del sistema operativo? ¿Qué mecanismos debería proporcionar el hardware? ¿Cómo podemos usarlos para resolver los problemas de concurrencia?

toma tres instrucciones: una para cargar el valor del contador de la memoria a un registro, otra para incrementarlo y otra para almacenarlo nuevamente en la memoria. Debido a que estas tres instrucciones no se ejecutan atómicamente (todas a la vez), pueden suceder cosas extrañas. Es este problema de concurrencia el que abordaremos con gran detalle en la segunda parte de este libro.

## 2.4 Persistencia

El tercer tema principal del curso es la perseverancia. En la memoria del sistema, los datos se pueden perder fácilmente, ya que dispositivos como la DRAM almacenan valores de forma volátil ; Cuando se corta la energía o el sistema falla, se pierden todos los datos de la memoria. Por tanto, necesitamos hardware y software para poder almacenar datos de forma persistente; Por lo tanto, dicho almacenamiento es fundamental para cualquier sistema, ya que los usuarios se preocupan mucho por sus datos.

El hardware viene en forma de algún tipo de entrada/salida o dispositivo de E/S ; En los sistemas modernos, un disco duro es un depósito común de información de larga duración, aunque las unidades de estado sólido (SSD) también están avanzando en este campo.

El software del sistema operativo que normalmente administra el disco se llama sistema de archivos; por lo tanto, es responsable de almacenar cualquier archivo que el usuario cree de manera confiable y eficiente en los discos del sistema.

A diferencia de las abstracciones proporcionadas por el sistema operativo para la CPU y la memoria, el sistema operativo no crea un disco virtualizado privado para cada aplicación. Más bien, se supone que muchas veces los usuarios querrán compartir información contenida en archivos. Por ejemplo, al escribir un programa en C, primero puede utilizar un editor (por ejemplo, Emacs<sup>7</sup>) para crear y editar el archivo C (emacs -nw main.c). Una vez hecho esto, puedes usar el compilador para convertir el código fuente en un ejecutable (por ejemplo, gcc -o main main.c). Cuando haya terminado, puede ejecutar el nuevo ejecutable (por ejemplo, ./main). De este modo, puede ver cómo se comparten los archivos entre diferentes procesos. Primero, Emacs crea un archivo que sirve como entrada para el compilador; el compilador usa ese archivo de entrada para crear un nuevo archivo ejecutable (en muchos pasos; tome un curso de compilador para obtener más detalles); finalmente, se ejecuta el nuevo ejecutable. ¡Y así nace un nuevo programa!

---

<sup>7</sup>Deberías utilizar Emacs. Si está utilizando vi, probablemente haya algún problema con tú. Si estás usando algo que no es un editor de código real, es aún peor.

```

1 #incluir <stdio.h>
2 #incluir <unistd.h>
3 #incluir <afirmar.h>
4 #incluir <fcntl.h>
5 #incluir <sys/types.h>
6
7 int principal(int argc, char *argv[]) {
8     int fd = open("/tmp/file", O_WRONLY|O_CREAT|O_TRUNC,
9                 S_IRWXU);
10    afirmar(fd > -1);
11    int rc = write(fd, "hola mundo\n", 13);
12    afirmar(rc == 13);
13    cerrar(fd);
14    devolver 0;
15 }

```

Figura 2.6: Un programa que realiza E/S (io.c)

Para entender esto mejor, veamos algo de código. La figura 2.6 presenta código para crear un archivo (`/tmp/file`) que contiene la cadena "hola mundo".

Para realizar esta tarea, el programa realiza tres llamadas al sistema operativo. La primera, una llamada a `open()`, abre el archivo y lo crea; el segundo, `write()`, escribe algunos datos en el archivo; el tercero, `close()`, simplemente cierra el archivo, indicando así que el programa no escribirá más datos al mismo. Estas llamadas al sistema se enrutan a la parte del sistema operativo llamada sistema de archivos, que luego maneja las solicitudes y devuelve algún tipo de código de error para el usuario.

Quizás se pregunte qué hace el sistema operativo para escribir al disco. Te lo mostraríamos pero tendrías que prometer cerrar tu ojos primero; es así de desagradable. El sistema de archivos tiene que hacer bastante trabajo: primero averiguando en qué parte del disco residirán estos nuevos datos y luego realizando un seguimiento de ellos en varias estructuras que mantiene el sistema de archivos. Haciéndolo requiere emitir solicitudes de E/S al dispositivo de almacenamiento subyacente, ya sea leer estructuras existentes o actualizarlas (escribir las). Como sabe cualquiera que haya escrito un controlador de dispositivo<sup>8</sup>, lograr que un dispositivo haga algo en su nombre es un proceso complejo y detallado. Requiere un conocimiento profundo de la interfaz del dispositivo de bajo nivel y su semántica exacta. Afortunadamente, el OS proporciona una forma estándar y sencilla de acceder a los dispositivos a través de sus llamadas al sistema. Por lo tanto, el sistema operativo a veces se considera una biblioteca estándar.

Por supuesto, hay muchos más detalles sobre cómo se accede a los dispositivos, y cómo los sistemas de archivos gestionan los datos de forma persistente sobre dichos dispositivos. Para razones de rendimiento, la mayoría de los sistemas de archivos primero retrasan dichas escrituras por un tiempo, con la esperanza de agruparlos en grupos más grandes. Para manejar los problemas de fallas del sistema durante las escrituras, la mayoría de los sistemas de archivos incorporan algún tipo de protocolo de escritura complejo, como llevar un diario o copiar sobre escritura, cuidadosamente

---

<sup>8</sup>Un controlador de dispositivo es un código en el sistema operativo que sabe cómo manejar un dispositivo específico. Hablaremos más sobre dispositivos y controladores de dispositivos más adelante.

### EL FUNDAMENTO DEL PROBLEMA:

#### CÓMO ALMACENAR DATOS DE FORMA PERSISTENTE

El sistema de archivos es la parte del sistema operativo encargada de gestionar los datos persistentes. ¿Qué técnicas se necesitan para hacerlo correctamente? ¿Qué mecanismos y ¿Se requieren políticas para hacerlo con alto desempeño? ¿Cómo es la confiabilidad? logrado, frente a fallas en hardware y software?

ordenar escrituras en el disco para garantizar que si ocurre una falla durante la escritura secuencia, el sistema puede recuperarse a un estado razonable posteriormente. para hacer Diferentes operaciones comunes eficientes, los sistemas de archivos emplean muchas estructuras de datos y métodos de acceso diferentes, desde listas simples hasta árboles b complejos. Si todo esto aún no tiene sentido, ¡bien! estaremos hablando de Todo esto un poco más en la tercera parte de este libro sobre la persistencia, donde discutiremos los dispositivos y E/S en general, y luego discos, RAID, y sistemas de archivos con gran detalle.

## 2.5 Objetivos de diseño

Ahora ya tienes una idea de lo que realmente hace un sistema operativo: toma recursos físicos , como una CPU, memoria o disco, y los virtualiza . Él maneja problemas difíciles y complicados relacionados con la concurrencia. Y almacena archivos persistentemente, haciéndolos así seguros a largo plazo. Dado que nosotros queremos construir un sistema de este tipo, queremos tener algunos objetivos en mente para ayudar centrar nuestro diseño e implementación y hacer concesiones según sea necesario; Encontrar el conjunto correcto de compensaciones es clave para construir sistemas.

Uno de los objetivos más básicos es construir algunas abstracciones para para que el sistema sea cómodo y fácil de usar. Las abstracciones son fundamentales para todo lo que hacemos en informática. La abstracción hace Es posible escribir un programa grande dividiéndolo en partes pequeñas y comprensibles, escribir dicho programa en un lenguaje de alto nivel como do<sup>9</sup> sin pensar en ensamblador, escribir código en ensamblador sin pensar en puertas lógicas y construir un procesador a partir de puertas sin Pensando demasiado en transistores. La abstracción es tan fundamental que a veces olvidamos su importancia, pero aquí no lo haremos; Por lo tanto, en cada sección, discutiremos algunas de las principales abstracciones que se han desarrollado. con el tiempo, brindándole una manera de pensar en partes del sistema operativo.

Uno de los objetivos al diseñar e implementar un sistema operativo es proporcionar un alto rendimiento; Otra forma de decir que nuestro objetivo es minimizar los gastos generales del sistema operativo. Virtualización y simplificación del sistema. valen la pena, pero no a cualquier precio; por lo tanto, debemos esforzarnos por ofrecer virtualización y otras funciones del sistema operativo sin gastos generales excesivos.

---

<sup>9</sup> Algunos de ustedes podrían oponerse a llamar a C un lenguaje de alto nivel. Recuerda que este es un sistema operativo. Por supuesto, sin embargo, estamos felices de no tener que codificar en ensamblador todo el tiempo.

Estos gastos generales surgen de diversas formas: tiempo extra (más instrucciones) y espacio extra (en memoria o en disco). Buscaremos soluciones que minimicen uno, el otro o ambos, si es posible. Sin embargo, la perfección no siempre es alcanzable, algo que aprenderemos a notar y (cuando sea apropiado) a tolerar.

Otro objetivo será brindar protección entre aplicaciones, así como entre el sistema operativo y las aplicaciones. Debido a que deseamos permitir que se ejecuten muchos programas al mismo tiempo, queremos asegurarnos de que el mal comportamiento malicioso o accidental de uno no dañe a otros; Ciertamente no queremos que una aplicación pueda dañar el sistema operativo (ya que eso afectaría a todos los programas que se ejecutan en el sistema). La protección está en el centro de uno de los principios fundamentales que subyacen a un sistema operativo, que es el del aislamiento; Aislar los procesos entre sí es la clave para la protección y, por lo tanto, es la base de gran parte de lo que debe hacer un sistema operativo.

El sistema operativo también debe funcionar sin parar; cuando falla, todas las aplicaciones que se ejecutan en el sistema también fallan. Debido a esta dependencia, los sistemas operativos a menudo se esfuerzan por proporcionar un alto grado de confiabilidad. A medida que los sistemas operativos se vuelven cada vez más complejos (a veces contienen millones de líneas de código), construir un sistema operativo confiable es todo un desafío y, de hecho, gran parte de la investigación en curso en este campo (incluidos algunos de nuestros propios trabajos) [BS+09, SS+10] se centra exactamente en este problema.

Otros objetivos tienen sentido: la eficiencia energética es importante en nuestro mundo cada vez más verde; la seguridad (en realidad, una extensión de la protección) contra aplicaciones maliciosas es fundamental, especialmente en estos tiempos de alta red; la movilidad es cada vez más importante a medida que los sistemas operativos se ejecutan en dispositivos cada vez más pequeños. Dependiendo de cómo se use el sistema, el sistema operativo tendrá diferentes objetivos y, por lo tanto, probablemente se implementará al menos de maneras ligeramente diferentes. Sin embargo, como veremos, muchos de los principios que presentaremos sobre cómo construir un sistema operativo son útiles en una variedad de dispositivos diferentes.

## 2.6 Un poco de historia

Antes de cerrar esta introducción, presentemos una breve historia de cómo se desarrollaron los sistemas operativos. Como cualquier sistema construido por humanos, las buenas ideas se acumularon en los sistemas operativos con el tiempo, a medida que los ingenieros aprendieron qué era importante en su diseño. Aquí analizamos algunos de los principales avances. Para un tratamiento más completo, consulte la excelente historia de los sistemas operativos de Brinch Hansen [BH00].

Los primeros sistemas operativos: sólo bibliotecas API

En principio, el sistema operativo no hacía demasiado. Básicamente, era sólo un conjunto de bibliotecas de funciones de uso común; por ejemplo, en lugar de que cada programador del sistema escriba código de manejo de E/S de bajo nivel, el "SO" proporcionaría dichas API y, por lo tanto, haría la vida más fácil para el desarrollador.

Por lo general, en estos viejos sistemas mainframe, se ejecutaba un programa a la vez, controlado por un operador humano. Gran parte de lo que cree que haría un sistema operativo moderno (por ejemplo, decidir en qué orden ejecutar los trabajos) fue realizado por este operador. Si fuera un desarrollador inteligente, sería amable con este operador para que pudiera mover su trabajo al frente de la cola.

Este modo de computación se conocía como procesamiento por lotes , ya que el operador configuraba una serie de trabajos y luego los ejecutaba en un "lote". Las computadoras, a partir de ese momento, no se usaban de manera interactiva debido al costo: simplemente era demasiado costoso permitir que un usuario se sentara frente a la computadora y la usara, ya que la mayor parte del tiempo simplemente permanecería inactiva. costando a la instalación cientos de miles de dólares por hora [BH00].

### Más allá de las bibliotecas: protección

Al ir más allá de ser una simple biblioteca de servicios de uso común, los sistemas operativos asumieron un papel más central en la gestión de máquinas. Un aspecto importante de esto fue darse cuenta de que el código ejecutado en nombre del sistema operativo era especial; tenía control de los dispositivos y por lo tanto debía ser tratado de manera diferente que el código de aplicación normal. ¿Por qué es esto? Bueno, imagínese si permitiera que cualquier aplicación leyera desde cualquier lugar del disco; la noción de privacidad desaparece, ya que cualquier programa podría leer cualquier archivo. Por lo tanto, implementar un sistema de archivos (para administrar sus archivos) como una biblioteca tiene poco sentido. En cambio, se necesitaba algo más.

Así, se inventó la idea de una llamada al sistema , iniciada por el sistema informático Atlas [K+61,L78]. En lugar de proporcionar rutinas del sistema operativo como una biblioteca (donde simplemente se realiza una llamada a un procedimiento para acceder a ellas), la idea aquí era agregar un par especial de instrucciones de hardware y estado del hardware para hacer que la transición al sistema operativo sea más formal y controlada. proceso.

La diferencia clave entre una llamada al sistema y una llamada a procedimiento es que una llamada al sistema transfiere el control (es decir, salta) al sistema operativo y al mismo tiempo eleva el nivel de privilegio del hardware. Las aplicaciones de usuario se ejecutan en lo que se conoce como modo de usuario, lo que significa que el hardware restringe lo que las aplicaciones pueden hacer; por ejemplo, una aplicación que se ejecuta en modo de usuario normalmente no puede iniciar una solicitud de E/S al disco, acceder a ninguna página de memoria física o enviar un paquete en la red. Cuando se inicia una llamada al sistema (generalmente a través de una instrucción de hardware especial llamada trampa), el hardware transfiere el control a un controlador de trampas preespecificado (que el sistema operativo configuró previamente) y simultáneamente eleva el nivel de privilegio al modo kernel. En modo kernel, el sistema operativo tiene acceso completo al hardware del sistema y, por lo tanto, puede hacer cosas como iniciar una solicitud de E/S o poner más memoria a disposición de un programa. Cuando el sistema operativo termina de atender la solicitud, devuelve el control al usuario a través de una instrucción especial de retorno desde trampa , que vuelve al modo de usuario y simultáneamente devuelve el control al lugar donde lo dejó la aplicación.

### La era de la multiprogramación Donde

realmente despegaron los sistemas operativos fue en la era de la informática más allá del mainframe, la de la minicomputadora. Las máquinas clásicas como la familia PDP de Digital Equipment hicieron que las computadoras fueran mucho más asequibles; por lo tanto, en lugar de tener una computadora central por organización grande, ahora un grupo más pequeño de personas dentro de una organización probablemente podría tener su propia computadora. No es sorprendente que uno de los principales impactos de esta caída en los costos fuera un aumento en la actividad de los desarrolladores; Más personas inteligentes pusieron sus manos en las computadoras y, por lo tanto, hicieron que los sistemas informáticos hicieran cosas más interesantes y hermosas.

En particular, la multiprogramación se volvió común debido al deseo de hacer un mejor uso de los recursos de las máquinas. En lugar de simplemente ejecutar un trabajo a la vez, el sistema operativo cargaría varios trabajos en la memoria y cambiaría rápidamente entre ellos, mejorando así la utilización de la CPU. Este cambio fue particularmente importante porque los dispositivos de E/S eran lentos; tener un programa esperando en la CPU mientras se reparaba su E/S era una pérdida de tiempo de la CPU. En lugar de eso, ¿por qué no cambiar a otro trabajo y ejecutarlo por un tiempo?

El deseo de apoyar la multiprogramación y la superposición en presencia de E/S e interrupciones forzaron la innovación en el desarrollo conceptual de los sistemas operativos en varias direcciones. Cuestiones como la protección de la memoria adquirieron importancia; No quisieramos que un programa pudiera acceder a la memoria de otro programa. También fue fundamental comprender cómo abordar los problemas de concurrencia introducidos por la multiprogramación; asegurarse de que el sistema operativo se comporte correctamente a pesar de la presencia de interrupciones es un gran desafío. Estudiaremos estos temas y temas relacionados más adelante en el libro.

Uno de los mayores avances prácticos de la época fue la introducción del sistema operativo UNIX , principalmente gracias a Ken Thompson (y Dennis Ritchie) de Bell Labs (sí, la compañía telefónica). UNIX tomó muchas buenas ideas de diferentes sistemas operativos (particularmente de Multics [O72], y algunas de sistemas como TENEX [B+72] y Berkeley Time-Sharing System [S+68]), pero las hizo más simples y fáciles de usar. . Pronto, este equipo estaba enviando cintas que contenían el código fuente UNIX a personas de todo el mundo, muchas de las cuales luego se involucraron y agregaron ellas mismas al sistema; consulte el aparte (página siguiente) para obtener más detalles<sup>10</sup> .

### La era moderna

Más allá de la minicomputadora surgió un nuevo tipo de máquina, más barata, más rápida y para las masas: la computadora personal o PC , como la llamamos hoy. Liderada por las primeras máquinas de Apple (por ejemplo, la Apple II) y la IBM PC, esta nueva generación de máquinas pronto se convertiría en la fuerza dominante en la informática.

---

<sup>10</sup>Usaremos apartes y otros cuadros de texto relacionados para llamar la atención sobre varios elementos que no se ajustan del todo al flujo principal del texto. A veces, incluso los usamos solo para hacer una broma, porque ¿por qué no divertirnos un poco en el camino? Si, muchos de los chistes son malos.

#### APARTE: LA IMPORTANCIA DE UNIX

Es difícil exagerar la importancia de UNIX en la historia de los sistemas operativos. Influenciado por sistemas anteriores (en particular, el famoso sistema Multics del MIT), UNIX reunió muchas ideas geniales y creó un sistema que era a la vez simple y poderoso.

Detrás del UNIX original de "Bell Labs" estaba el principio unificador de crear pequeños programas potentes que pudieran conectarse entre sí para formar flujos de trabajo más grandes. El shell, donde se escriben comandos, proporcionaba primitivas como canalizaciones para permitir dicha programación de metanivel y, por lo tanto, resultó fácil encadenar programas para realizar una tarea más grande. Por ejemplo, para buscar líneas de un archivo de texto que tengan la palabra "foo" y luego contar cuántas de esas líneas existen, escribiría: grep foo file.txt|wc -l, usando así grep y wc (recuento de palabras) programas para lograr su tarea.

El entorno UNIX era amigable tanto para programadores como para desarrolladores y también proporcionaba un compilador para el nuevo lenguaje de programación C. Facilitar a los programadores escribir sus propios programas, así como compartirlos, hizo que UNIX se hiciera enormemente popular. Y probablemente ayudó mucho que los autores distribuyeran copias gratuitas a cualquiera que las solicitara, una de las primeras formas de software de código abierto.

También fue de importancia crítica la accesibilidad y legibilidad del código. Tener un kernel pequeño y hermoso escrito en C invitó a otros a jugar con el kernel, agregando características nuevas y interesantes. Por ejemplo, un grupo emprendedor en Berkeley, dirigido por Bill Joy, creó una maravillosa distribución (Berkeley Systems Distribution, o BSD) que tenía algunos subsistemas avanzados de memoria virtual, sistema de archivos y redes. Más tarde, Joy cofundó Sun Microsystems.

Desafortunadamente, la difusión de UNIX se ralentizó un poco a medida que las empresas intentaron afirmar su propiedad y beneficiarse de él, un resultado desafortunado (pero común) de la participación de abogados. Muchas empresas tenían sus propias variantes: SunOS de Sun Microsystems, AIX de IBM, HPUX (también conocido como "H-Pucks") de HP e IRIX de SGI. Las disputas legales entre AT&T/Bell Labs y estos otros jugadores arrojaron una nube oscura sobre UNIX, y muchos se preguntaron si sobreviviría, especialmente cuando se introdujo Windows y se apoderó de gran parte del mercado de PC...

ya que su bajo costo permitía una máquina por escritorio en lugar de una minicomputadora compartida por grupo de trabajo.

Desafortunadamente, para los sistemas operativos, la PC representó al principio un gran salto hacia atrás, ya que los primeros sistemas olvidaron (o nunca supieron) las lecciones aprendidas en la era de las minicomputadoras. Por ejemplo, los primeros sistemas operativos como DOS (el sistema operativo de disco, de Microsoft) no pensaban que la protección de la memoria fuera importante; por lo tanto, una aplicación maliciosa (o quizás simplemente mal programada) podría garabatear toda la memoria.

### APARTE: Y LUEGO LLEGÓ LINUX

Afortunadamente para UNIX, un joven hacker finlandés llamado Linus Torvalds decidió escribir su propia versión de UNIX, que se basó en gran medida en el Principios e ideas detrás del sistema original, pero no del código. base, evitando así cuestiones de legalidad. Pidió ayuda a muchos otros en todo el mundo, aprovecharon las sofisticadas herramientas GNU que ya existía [G85], y pronto nació Linux (así como el moderno movimiento de software de código abierto).

A medida que llegó la era de Internet, la mayoría de las empresas (como Google, Amazon, Facebook y otros) eligieron ejecutar Linux, ya que era gratuito y podría modificarse fácilmente para satisfacer sus necesidades; de hecho, es difícil imaginar el éxito de estas nuevas empresas si tal sistema no existiera.

A medida que los teléfonos inteligentes se convirtieron en una plataforma dominante para el usuario, Linux descubrió También es un bastión allí (a través de Android), por muchas de las mismas razones. Y Steve Jobs tomó su entorno operativo NeXTStep basado en UNIX con a Apple, haciendo así que UNIX fuera popular en las computadoras de escritorio (aunque muchos los usuarios de la tecnología Apple probablemente ni siquiera sean conscientes de este hecho). De este modo UNIX sigue vivo y es más importante hoy que nunca. la informática Los dioses, si crees en ellos, deben ser agradecidos por este maravilloso descubrimiento. venir.

oria. Las primeras generaciones de Mac OS (v9 y anteriores) adoptaron un enfoque cooperativo para la programación de trabajos; así, un hilo que accidentalmente se atascó en un bucle infinito podría apoderarse de todo el sistema, forzando un reinicio. El dolorosa lista de características del sistema operativo que faltan en esta generación de sistemas es larga, demasiado tiempo para una discusión completa aquí.

Afortunadamente, después de algunos años de sufrimiento, las antiguas características de los sistemas operativos de las minicomputadoras comenzaron a llegar al escritorio. Por ejemplo, Mac OS X/macOS tiene UNIX en su núcleo, incluidos todos los características que uno esperaría de un sistema tan maduro. De manera similar, Windows ha adoptado muchas de las grandes ideas de la historia de la informática, empezando por Particularmente con Windows NT, un gran salto adelante en la tecnología del sistema operativo Microsoft. Incluso los teléfonos móviles actuales ejecutan sistemas operativos (como Linux) que se parecen mucho más a lo que hacía funcionar una minicomputadora en la década de 1970 que a lo que una PC funcionó en la década de 1980 (gracias a Dios); es bueno ver que el bien Las ideas desarrolladas en el apogeo del desarrollo de sistemas operativos han encontrado su camino. al mundo moderno. Aún mejor es que estas ideas continúan desarrollándose, proporcionando más funciones y haciendo que los sistemas modernos sean aún mejores. para usuarios y aplicaciones.

## 2.7 Resumen

Así, tenemos una introducción al sistema operativo. Los sistemas operativos actuales hacen que los sistemas sean relativamente fáciles de usar y prácticamente todos los sistemas operativos que utilizas hoy han sido influenciados por los desarrollos que discutiremos a lo largo del libro.

Desafortunadamente, debido a limitaciones de tiempo, hay varias partes de el sistema operativo que no cubriremos en el libro. Por ejemplo, hay una gran cantidad de código de red en el sistema operativo; Le dejamos a usted tomar la clase de trabajo en red para aprender más sobre eso. De manera similar, los dispositivos gráficos son particularmente importante; toma el curso de gráficos para ampliar tus conocimientos en esa dirección. Finalmente, algunos libros sobre sistemas operativos hablan muy bien tratar sobre seguridad; lo haremos en el sentido de que el sistema operativo debe proporcionar protección entre programas en ejecución y brindar a los usuarios la capacidad de proteger sus archivos, pero no profundizaremos en problemas de seguridad más profundos que uno podría encontrar en un curso de seguridad.

Sin embargo, hay muchos temas importantes que cubriremos, incluidos los conceptos básicos de virtualización de la CPU y la memoria, la concurrencia y persistencia a través de dispositivos y sistemas de archivos. ¡No te preocupes! Si bien hay un Hay mucho terreno por recorrer, la mayor parte es bastante fresco, y al final del camino, Tendrá una nueva apreciación de cómo funcionan realmente los sistemas informáticos. ¡Ahora manos a la obra!

## Referencias

- [BS+09] "Tolerar errores en el sistema de archivos con EnvyFS" por L. Bairavasundaram, S. Sundarara-man, A. Arpac-Dusseau, R. Arpac-Dusseau. USENIX '09, San Diego, CA, junio de 2009. Un artículo divertido sobre el uso de múltiples sistemas de archivos a la vez para tolerar un error en cualquiera de ellos.
- [BH00] "La evolución de los sistemas operativos" de P. Brinch Hansen. En 'Sistemas operativos clásicos: del procesamiento por lotes a sistemas distribuidos'. Springer-Verlag, Nueva York, 2000.
- Este ensayo proporciona una introducción a una maravillosa colección de artículos sobre sistemas de importancia histórica.
- [B+72] "TENEX, un sistema de tiempo compartido paginado para el PDP-10" por D. Bobrow, J. Burchfiel, D. Murphy, R. Tomlinson. CACM, Volumen 15, Número 3, marzo de 1972. TENEX tiene gran parte de la maquinaria que se encuentra en los sistemas operativos modernos; Lea más sobre esto para ver cuánta innovación ya existía a principios de la década de 1970.
- [B75] "El mes del hombre mítico" por F. Brooks. Addison-Wesley, 1975. Un texto clásico sobre ingeniería de software; bien vale la pena leerlo.
- [BOH10] "Sistemas informáticos: la perspectiva de un programador" por R. Bryant y D. O'Hallaron. Addison-Wesley, 2010. Otra gran introducción a cómo funcionan los sistemas informáticos. Se superpone un poco con este libro, por lo que si lo desea, puede omitir los últimos capítulos de ese libro o simplemente leerlos para obtener una perspectiva diferente sobre parte del mismo material. Después de todo, una buena manera de desarrollar su propio conocimiento es escuchar tantas otras perspectivas como sea posible y luego desarrollar su propia opinión y pensamientos sobre el tema. Ya sabes, ¡pensando!
- [G85] "El Manifiesto GNU" por R. Stallman. 1985. [www.gnu.org/gnu/manifesto.html](http://www.gnu.org/gnu/manifesto.html).
- Una gran parte del éxito de Linux fue sin duda la presencia de un excelente compilador, gcc, y otras piezas relevantes de software abierto, gracias al esfuerzo de GNU encabezado por Stallman. Stallman es un visionario en lo que respecta al código abierto, y este manifiesto explica por qué.
- [K+61] "Sistema de almacenamiento de un nivel" por 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. Sin embargo, este artículo no es la mejor lectura. Si solo leyeras uno, podrías probar la perspectiva histórica a continuación [L78].
- [L78] "El Manchester Mark I y el Atlas: una perspectiva histórica" por SH Lavington. Communications of the ACM, Volumen 21:1, enero de 1978. Un bonito fragmento de historia sobre el desarrollo inicial de los sistemas informáticos y los esfuerzos pioneros del Atlas. Por supuesto, uno podría volver atrás y leer los artículos del Atlas, pero este documento proporciona una excelente descripción general y agrega cierta perspectiva histórica.
- [O72] "El sistema Multics: un examen de su estructura" por Elliott Organick. MIT Press, 1972. Una excelente descripción general de Multics. Tantas buenas ideas y, sin embargo, era un sistema sobrediseñado, que apuntaba a demasiado y, por lo tanto, nunca funcionó realmente. Un ejemplo clásico de lo que Fred Brooks llamaría el "efecto del segundo sistema" [B75].
- [PP03] "Introducción a los sistemas informáticos: de bits y puertas a C y más allá" por Yale N. Patt, Sanjay J. Patel. McGraw-Hill, 2003. Uno de nuestros libros favoritos de introducción a los sistemas informáticos. Comienza en los transistores y llega hasta C; El material inicial es particularmente bueno.
- [RT74] "El sistema de tiempo compartido UNIX" por Dennis M. Ritchie, Ken Thompson. CACM, Volumen 17: 7, julio de 1974. Un gran resumen de UNIX escrito mientras se apoderaba del mundo de la informática, por las personas que lo escribieron.
- [S68] "Sistema de tiempo compartido SDS 940" de Scientific Data Systems. MANUAL TÉCNICO, SDS 90 11168, agosto de 1968. Sí, un manual técnico fue lo mejor que pudimos encontrar. Pero es fascinante leer estos documentos del antiguo sistema y ver cuánto ya existía a finales de los años 1960. Una de las mentes detrás del Sistema de Tiempo Compartido de Berkeley (que eventualmente se convirtió en el sistema SDS) fue Butler Lampson, quien más tarde ganó un premio Turing por sus contribuciones en sistemas.
- [SS+10] "Membrane: soporte del sistema operativo para sistemas de archivos reiniciables" por S. Sundarara-man, S. Subramanian, A. Rajimwale, R. Arpac-Dusseau, M. Swift. FAST '10, San José, CA, febrero de 2010. Lo mejor de escribir tus propios apuntes de clase: puedes anunciar tu propia investigación. Pero este documento en realidad es bastante interesante: cuando un sistema de archivos sufre un error y falla, Membrane lo reinicia automáticamente, todo sin que las aplicaciones ni el resto del sistema se vean afectados.

## Tarea

La mayoría (y eventualmente todos) los capítulos de este libro tienen secciones de tareas al final. Hacer estas tareas es importante, ya que cada una le permite a usted, el lector, adquirir más experiencia con los conceptos presentados en el capítulo.

Hay dos tipos de tareas. El primero se basa en la simulación. Una simulación de un sistema informático es simplemente un programa simple que pretende hacer algunas de las partes interesantes de lo que hace un sistema real y luego informar algunas métricas de salida para mostrar cómo se comporta el sistema. Por ejemplo, un simulador de disco duro podría tomar una serie de solicitudes, simular cuánto tiempo tardarían en ser atendidas por un disco duro con ciertas características de rendimiento y luego informar la latencia promedio de las solicitudes.

Lo bueno de las simulaciones es que te permiten explorar fácilmente cómo se comportan los sistemas sin la dificultad de ejecutar un sistema real. De hecho, incluso permiten crear sistemas que no pueden existir en el mundo real (por ejemplo, un disco duro con un rendimiento inimaginablemente rápido) y así ver el impacto potencial de las tecnologías futuras.

Por supuesto, las simulaciones no están exentas de desventajas. Por su propia naturaleza, las simulaciones son sólo aproximaciones de cómo se comporta un sistema real. Si se omite un aspecto importante del comportamiento en el mundo real, la simulación arrojará malos resultados. Por tanto, los resultados de una simulación siempre deben tratarse con cierta sospecha. Al final, lo que importa es cómo se comporta un sistema en el mundo real.

El segundo tipo de tarea requiere interacción con código del mundo real. Algunas de estas tareas se centran en la medición, mientras que otras simplemente requieren algo de desarrollo y experimentación a pequeña escala. Ambas son sólo pequeñas incursiones en el mundo más amplio al que debería adentrarse, que es cómo escribir código de sistemas en C en sistemas basados en UNIX. De hecho, se necesitan proyectos de mayor escala, que vayan más allá de estas tareas, para impulsarlos en esta dirección; por lo tanto, más allá de simplemente hacer las tareas, le recomendamos encarecidamente que realice proyectos para solidificar sus habilidades en sistemas. Consulte esta página (<https://github.com/remzi-arpacidusseau/ostep-projects>) para ver algunos proyectos.

Para hacer estas tareas, es probable que tenga que estar en una máquina basada en UNIX, ejecutando Linux, macOS o algún sistema similar. También debería tener instalado un compilador de C (por ejemplo, gcc), además de Python. También deberías saber cómo editar código en un editor de código real de algún tipo.

## La abstracción: el proceso

En este capítulo, analizamos una de las abstracciones más fundamentales que el sistema operativo proporciona a los usuarios: el proceso. La definición de proceso, informalmente, es bastante simple: es un programa en ejecución [V+65,BH70]. El programa en sí es algo sin vida: simplemente se queda ahí en el disco, un montón de instrucciones (y tal vez algunos datos estáticos), esperando entrar en acción. Es el sistema operativo el que toma estos bytes y los pone en ejecución, transformando el programa en algo útil.

Resulta que a menudo uno quiere ejecutar más de un programa a la vez; por ejemplo, considere su computadora de escritorio o portátil donde le gustaría ejecutar un navegador web, un programa de correo, un juego, un reproductor de música, etc. De hecho, un sistema típico puede aparentemente estar ejecutando decenas o incluso cientos de procesos al mismo tiempo. Esto hace que el sistema sea fácil de usar, ya que nunca es necesario preocuparse por si hay una CPU disponible; uno simplemente ejecuta programas. De ahí nuestro desafío:

EL MEJOR DEL PROBLEMA: ¿ CÓMO  
PROPORCIONAR LA ILUSIÓN DE MUCHAS CPU?

Aunque sólo hay unas pocas CPU físicas disponibles, ¿cómo puede  
¿El sistema operativo proporciona la ilusión de un suministro casi infinito de dichas CPU?

El sistema operativo crea esta ilusión virtualizando la CPU. Al ejecutar un proceso, luego detenerlo y ejecutar otro, y así sucesivamente, el sistema operativo puede promover la ilusión de que existen muchas CPU virtuales cuando en realidad sólo hay una CPU física (o varias). Esta técnica básica, conocida como tiempo compartido de la CPU, permite a los usuarios ejecutar tantos procesos simultáneos como deseen; el costo potencial es el rendimiento, ya que cada uno se ejecutará más lentamente si se deben compartir las CPU.

Para implementar la virtualización de la CPU, y para implementarla bien, el sistema operativo necesitará tanto maquinaria de bajo nivel como inteligencia de alto nivel. A los mecanismos de maquinaria de bajo nivel los llamamos ; Los mecanismos son métodos o protocolos de bajo nivel que implementan una funcionalidad necesaria. Por ejemplo, aprenderemos más adelante cómo implementar un contexto.

**CONSEJO: UTILICE EL TIEMPO COMPARTIDO (Y EL ESPACIO COMPARTIDO)**

El tiempo compartido es una técnica básica utilizada por un sistema operativo para compartir un recurso. Al permitir que el recurso sea utilizado durante un tiempo por una entidad, y luego por otro tiempo por otra, y así sucesivamente, el recurso en cuestión (por ejemplo, la CPU o un enlace de red) puede ser compartido por muchos. La contraparte del tiempo compartido es el espacio compartido, donde un recurso se divide (en el espacio) entre quienes desean utilizarlo. Por ejemplo, el espacio en disco es naturalmente un recurso de espacio compartido; Una vez que se asigna un bloque a un archivo, normalmente no se asigna a otro archivo hasta que el usuario elimina el archivo original.

switch, que le da al sistema operativo la capacidad de detener la ejecución de un programa y comenzar a ejecutar otro en una CPU determinada; Este mecanismo de tiempo compartido lo emplean todos los sistemas operativos modernos.

Además de estos mecanismos reside parte de la inteligencia del sistema operativo, en forma de políticas. Las políticas son algoritmos para tomar algún tipo de decisión dentro del sistema operativo. Por ejemplo, dada una cantidad de programas posibles para ejecutar en una CPU, ¿qué programa debería ejecutar el sistema operativo? Una política de programación en el sistema operativo tomará esta decisión, probablemente utilizando información histórica (por ejemplo, ¿qué programa se ha ejecutado más en el último minuto?), conocimiento de la carga de trabajo (por ejemplo, qué tipos de programas se ejecutan) y métricas de rendimiento. (por ejemplo, ¿está el sistema optimizando el rendimiento interactivo o el rendimiento?) para tomar su decisión.

#### 4.1 La abstracción: un proceso

La abstracción proporcionada por el sistema operativo de un programa en ejecución es algo que llamaremos proceso . Como dijimos anteriormente, un proceso es simplemente un programa en ejecución; En cualquier instante, podemos resumir un proceso haciendo un inventario de las diferentes partes del sistema al que accede o afecta durante el curso de su ejecución.

Para entender qué constituye un proceso, tenemos que entender el estado de su máquina: lo que un programa puede leer o actualizar cuando se está ejecutando.

En un momento dado, ¿qué partes de la máquina son importantes para la ejecución de este programa?

Un componente obvio del estado de la máquina que comprende un proceso es su memoria. Las instrucciones se encuentran en la memoria; los datos que el programa en ejecución lee y escribe también se encuentran en la memoria. Por tanto, la memoria a la que el proceso puede direccionar (llamada espacio de direcciones) es parte del proceso.

También forman parte del estado de la máquina del proceso los registros; Muchas instrucciones leen o actualizan registros explícitamente y, por lo tanto, claramente son importantes para la ejecución del proceso.

Tenga en cuenta que hay algunos registros particularmente especiales que forman parte de este estado de la máquina. Por ejemplo, el contador de programa (PC) (a veces llamado puntero de instrucción o IP) nos dice qué instrucción del programa se ejecutará a continuación; de manera similar, un puntero de pila y un marco asociado

**CONSEJO: POLÍTICAS Y MECANISMOS SEPARADOS**

En muchos sistemas operativos, un paradigma de diseño común es separar las políticas de alto nivel de sus mecanismos de bajo nivel [L+75]. Se puede pensar que el mecanismo proporciona la respuesta a una pregunta sobre cómoacerca de un sistema; por ejemplo, ¿cómo realiza un sistema operativo un cambio de contexto? La política proporciona la respuesta a una pregunta: ¿qué? por ejemplo, ¿qué proceso debería ejecutar el sistema operativo en este momento? Separar los dos permite cambiar fácilmente las políticas sin tener que repensar el mecanismo y, por lo tanto, es una forma de modularidad, un principio general de diseño de software.

Los punteros se utilizan para administrar la pila de parámetros de función, variables locales y direcciones de retorno.

Finalmente, los programas también suelen acceder a dispositivos de almacenamiento persistentes. Dicha información de E/S podría incluir una lista de los archivos que el proceso tiene abiertos actualmente.

## 4.2 API de proceso

Aunque aplazamos la discusión sobre una API de proceso real hasta un capítulo posterior, aquí primero damos una idea de lo que debe incluirse en cualquier interfaz de un sistema operativo. Estas API, de alguna forma, están disponibles en cualquier sistema operativo moderno.

- Crear: Un sistema operativo debe incluir algún método para crear nuevos procesos. Cuando escribe un comando en el shell o hace doble clic en el ícono de una aplicación, se invoca el sistema operativo para crear un nuevo proceso para ejecutar el programa que ha indicado.
- Destruir: así como existe una interfaz para la creación de procesos, los sistemas también proporcionan una interfaz para destruir procesos por la fuerza. Por supuesto, muchos procesos se ejecutarán y saldrán solos cuando se completen; Sin embargo, cuando no lo hacen, es posible que el usuario desee matarlos y, por lo tanto, una interfaz para detener un proceso fuera de control es bastante útil.
- Esperar: A veces es útil esperar a que un proceso deje de ejecutarse; por tanto, a menudo se proporciona algún tipo de interfaz de espera.
- Control varios: además de matar o esperar un proceso, a veces hay otros controles posibles. Por ejemplo, la mayoría de los sistemas operativos proporcionan algún tipo de método para suspender un proceso (detener su ejecución por un tiempo) y luego reanudarlo (continuar ejecutándose).
- Estado: normalmente también hay interfaces para obtener información sobre el estado de un proceso, como por ejemplo cuánto tiempo se ha ejecutado o en qué estado se encuentra.

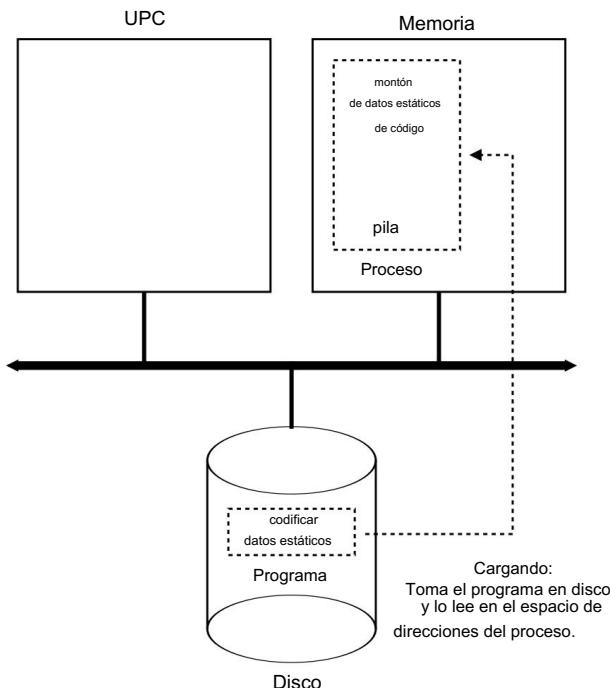


Figura 4.1: Cargando: Del programa al proceso

### 4.3 Creación de procesos: un poco más de detalle

Un misterio que deberíamos desenmascarar un poco es cómo se transforman los programas en procesos. Específicamente, ¿cómo hace el sistema operativo para que un programa esté en funcionamiento? ¿Cómo funciona realmente la creación de procesos?

Lo primero que debe hacer el sistema operativo para ejecutar un programa es cargar su código y cualquier dato estático (por ejemplo, variables inicializadas) en la memoria, en el espacio de direcciones del proceso. Inicialmente, los programas residen en el disco (o, en algunos sistemas modernos, en SSD basados en flash) en algún tipo de formato ejecutable; por lo tanto, el proceso de cargar un programa y datos estáticos en la memoria requiere que el sistema operativo lea esos bytes del disco y los coloque en algún lugar de la memoria (como se muestra en la Figura 4.1).

En los primeros (o simples) sistemas operativos, el proceso de carga se realiza con entusiasmo, es decir, todo a la vez antes de ejecutar el programa; Los sistemas operativos modernos realizan el proceso de forma perezosa, es decir, cargando fragmentos de código o datos sólo cuando son necesarios durante la ejecución del programa. Para comprender realmente cómo funciona la carga diferida de fragmentos de código y datos, deberá comprender más sobre

la maquinaria de paginación e intercambio, temas que cubriremos en el futuro cuando analicemos la virtualización de la memoria. Por ahora, recuerde que antes de ejecutar cualquier cosa, el sistema operativo claramente debe trabajar un poco para llevar los bits importantes del programa del disco a la memoria.

Una vez que el código y los datos estáticos se cargan en la memoria, hay algunas otras cosas que el sistema operativo debe hacer antes de ejecutar el proceso. Se debe asignar algo de memoria para la pila de tiempo de ejecución del programa (o simplemente para la pila). Como probablemente ya sabrá, los programas en C utilizan la pila para variables locales, parámetros de funciones y direcciones de retorno; el sistema operativo asigna esta memoria y se la da al proceso. Es probable que el sistema operativo también inicialice la pila con argumentos; específicamente, completará los parámetros de la función main() , es decir, argc y la matriz argv .

El sistema operativo también puede asignar algo de memoria para el montón del programa . En los programas C, el montón se utiliza para datos asignados dinámicamente solicitados explícitamente; los programas solicitan dicho espacio llamando a malloc() y lo liberan explícitamente llamando a free(). El montón es necesario para estructuras de datos como listas vinculadas, tablas hash, árboles y otras estructuras de datos interesantes. El montón será pequeño al principio; A medida que el programa se ejecuta y solicita más memoria a través de la API de la biblioteca malloc() , el sistema operativo puede involucrarse y asignar más memoria al proceso para ayudar a satisfacer dichas llamadas.

El sistema operativo también realizará otras tareas de inicialización, particularmente las relacionadas con la entrada/salida (E/S). Por ejemplo, en los sistemas UNIX , cada proceso tiene de forma predeterminada tres descriptores de archivos abiertos, para entrada, salida y error estándar; Estos descriptores permiten a los programas leer fácilmente la entrada desde el terminal e imprimir la salida en la pantalla. Aprendaremos más sobre E/S, descriptores de archivos y similares en la tercera parte del libro sobre persistencia.

Al cargar el código y los datos estáticos en la memoria, crear e inicializar una pila y realizar otros trabajos relacionados con la configuración de E/S, el sistema operativo ahora (finalmente) ha preparado el escenario para la ejecución del programa. Por lo tanto, tiene una última tarea: iniciar la ejecución del programa en el punto de entrada, es decir, main(). Al saltar a la rutina main() (a través de un mecanismo especializado que discutiremos en el próximo capítulo), el sistema operativo transfiere el control de la CPU al proceso recién creado y, por lo tanto, el programa comienza su ejecución.

#### 4.4 Estados del proceso

Ahora que tenemos una idea de qué es un proceso (aunque continuaremos perfeccionando esta noción) y (aproximadamente) cómo se crea, hablemos de los diferentes estados en los que puede encontrarse un proceso en un momento dado. La noción de que un proceso puede estar en uno de estos estados surgió en los primeros sistemas informáticos [DV66,V+65]. En una vista simplificada, un proceso puede estar en uno de tres estados:

- En ejecución: en el estado de ejecución, un proceso se está ejecutando en un procesador. Esto significa que está ejecutando instrucciones.
- Listo: en el estado listo, un proceso está listo para ejecutarse pero por alguna razón el sistema operativo ha decidido no ejecutarlo en ese momento.

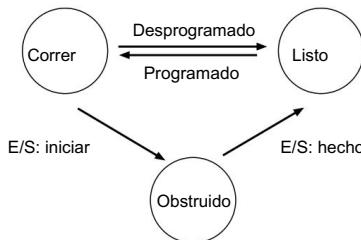


Figura 4.2: Proceso: Transiciones de Estado

- Bloqueado: en el estado bloqueado, un proceso ha realizado algún tipo de operación que lo hace no estar listo para ejecutarse hasta algún otro evento tiene lugar. Un ejemplo común: cuando un proceso inicia una E/S solicitud a un disco, éste se bloquea y, por lo tanto, algún otro proceso Puede utilizar el procesador.

Si mapeáramos estos estados en un gráfico, llegaríamos al diagrama de la figura 4.2. Como se puede ver en el diagrama, un proceso puede ser movido entre los estados listo y en ejecución a discreción del sistema operativo. Pasar de listo a en ejecución significa que el proceso ha sido programado; pasar de en ejecución a listo significa que el proceso ha sido desprogramado. Una vez que un proceso se ha bloqueado (por ejemplo, al iniciar una operación de E/S), el sistema operativo lo mantendrá así hasta que ocurra algún evento (p. ej., finalización de E/S); en ese punto, el proceso pasa nuevamente al estado listo (y potencialmente volver a ejecutarse inmediatamente, si el sistema operativo así lo decide).

Veamos un ejemplo de cómo dos procesos podrían pasar a través de algunos de estos estados. Primero, imagine dos procesos ejecutándose, cada uno de los cuales solo usan la CPU (no hacen E/S). En este caso, un rastro del estado de cada uno. El proceso podría verse así (Figura 4.3).

Tiempo	Proceso0	Proceso1	Notas
1	Listo para correr		
2	Listo para correr		
3	Listo para correr		
4	Ejecutando Ready Process0 ahora hecho		
5	-	Correr	
6	-	Correr	
7	-	Correr	
8	-	Ejecutando el Proceso 1 ahora hecho	

Figura 4.3: Estado del proceso de seguimiento: solo CPU

Tiempo	Proceso0	Proceso1	Notas
1	Listo para correr		
2	Listo para correr		
3	La ejecución del proceso Ready0 inicia la E/S		
4	Bloqueado El proceso en ejecución 0 está bloqueado,		
5	Bloqueado en ejecución para que se ejecute el Proceso1		
6	Ejecución bloqueada		
7	Listo para correr		E/S realizada
8	Listo, el proceso en ejecución 1 ya está listo		
9	Correr	-	
10 corriendo	-		Proceso0 ya hecho

Figura 4.4: Estado del proceso de seguimiento: CPU y E/S

En el siguiente ejemplo, el primer proceso emite una E/S después de ejecutarse durante a veces. En ese punto, el proceso se bloquea, dándole al otro proceso una oportunidad de correr. La figura 4.4 muestra un rastro de este escenario.

Más específicamente, Process0 inicia una E/S y se bloquea esperando a que se complete; Los procesos se bloquean, por ejemplo, al leer desde un disco o al esperar un paquete de una red. El sistema operativo reconoce que el Proceso0 no está utilizando la CPU y comienza a ejecutar el Proceso1. Mientras El proceso1 se está ejecutando, la E/S se completa y el proceso0 vuelve a estar listo. Finalmente, el Proceso1 finaliza, el Proceso0 se ejecuta y luego finaliza.

Tenga en cuenta que hay muchas decisiones que el sistema operativo debe tomar, incluso en este ejemplo sencillo. Primero, el sistema tuvo que decidir ejecutar el Proceso1 mientras Process0 emitió una E/S; Al hacerlo, se mejora la utilización de recursos al mantener ocupada la CPU. En segundo lugar, el sistema decidió no volver a Process0 cuando se completó su E/S; No está claro si es una buena decisión o no. ¿Qué opinas? Este tipo de decisiones las toma el Programador del sistema operativo , un tema que discutiremos en algunos capítulos en el futuro.

## 4.5 Estructuras de datos

El sistema operativo es un programa y, como cualquier programa, tiene algunas estructuras de datos clave que rastrean diversos datos relevantes. Para rastrear el estado de cada proceso, por ejemplo, el sistema operativo probablemente mantendrá algún tipo de lista de procesos para todos los procesos que están listos y alguna información adicional para rastrear qué proceso se está ejecutando actualmente. El sistema operativo también debe realizar un seguimiento, de alguna manera, procesos bloqueados; Cuando se completa un evento de E/S, el sistema operativo debe asegurarse de activar el proceso correcto y prepararlo para ejecutarse nuevamente.

La Figura 4.5 muestra qué tipo de información necesita rastrear un sistema operativo cada proceso en el kernel xv6 [CK+08]. Existen estructuras de proceso similares en sistemas operativos "reales" como Linux, Mac OS X o Windows; mirar Levántelos y vea cuánto más complejos son.

En la figura, puede ver un par de datos importantes que el sistema operativo rastrea sobre un proceso. El contexto del registro se mantendrá, durante un

```

// los registros xv6 se guardarán y restaurarán
// para detener y posteriormente reiniciar un proceso
contexto de estructura {
    int eip;
    int esp;
    int ebx;
    int ecx;
    int edx;
    int esi;
    edición internacional;
    int epp;
};

// los diferentes estados en los que puede estar un proceso
enum proc_state { NO UTILIZADO, EMBRIÓN, DURMIENDO,
    EJECUTABLE, CORRIENDO, ZOMBIE };

// la información que xv6 rastrea sobre cada proceso
// incluyendo su contexto y estado de registro
proceso de estructura {
    carbón *mem;                                // Inicio de la memoria del proceso
    uint tamaño;                                 // Tamaño de la memoria del proceso
    char *kpila;                                // Parte inferior de la pila del kernel
                                                // para este proceso
    enumeración proc_state estado;              // estado del proceso
    intpid; // ID del proceso
    estructura proc *padre; // Si !zero,           // proceso padre
    vacío *chan; // Si !                         durmiendo en chan
    muerto; archivo dzero, ha sido asesinado
    estructura *ofile[NOFILE]; // Abrir archivos
    inodo de estructura *cwd;                   // directorio actual
    estructura contexto contexto; estructura   // Cambia aquí para ejecutar el proceso
    trampa *tf,                                  // Trampa para el
                                                // interrupción actual
};


```

Figura 4.5: La estructura del proceso xv6

proceso detenido, el contenido de sus registros. Cuando se detiene un proceso, sus registros se guardarán en esta ubicación de memoria; Al restaurar estos registros (es decir, colocar sus valores nuevamente en los registros físicos reales), el sistema operativo puede reanudar la ejecución del proceso. Aprenderemos más sobre esta técnica. conocido como cambio de contexto en futuros capítulos.

También puede ver en la figura que hay otros estados en los que puede estar un proceso, además de en ejecución, listo y bloqueado. A veces un sistema tendrá un estado inicial en el que se encuentra el proceso cuando se está creando. Además, un proceso podría colocarse en un estado final del que ha salido pero

**APARTE: ESTRUCTURA DE DATOS : LA LISTA DE**

PROCESOS Los sistemas operativos están repletos de varias estructuras de datos importantes que analizaremos en estas notas. La lista de procesos (también llamada lista de tareas) es la primera estructura de este tipo. Es uno de los más simples, pero ciertamente cualquier sistema operativo que tenga la capacidad de ejecutar múltiples programas a la vez tendrá algo parecido a esta estructura para realizar un seguimiento de todos los programas en ejecución en el sistema. A veces la gente se refiere a la estructura individual que almacena información sobre un proceso como Bloque de control de procesos (PCB), una forma elegante de hablar de una estructura C que contiene información sobre cada proceso (a veces también llamada descriptor de proceso).

aún no se ha limpiado (en sistemas basados en UNIX, esto se llama estado zombie<sup>1</sup>). Este estado final puede ser útil ya que permite que otros procesos (generalmente el padre que creó el proceso) examinen el código de retorno del proceso y vean si el proceso recién finalizado se ejecutó exitosamente (generalmente, los programas devuelven cero en sistemas basados en UNIX cuando han realizado una tarea con éxito y, en caso contrario, un valor distinto de cero). Cuando termine, el padre realizará una última llamada (por ejemplo, esperar()) para esperar a que se complete el hijo y también para indicarle al sistema operativo que puede limpiar cualquier estructura de datos relevante que se refiera al ahora. proceso

## 4.6 Resumen

Hemos introducido la abstracción más básica del sistema operativo: el proceso. Simplemente se ve como un programa en ejecución. Con esta visión conceptual en mente, pasaremos ahora al meollo de la cuestión: los mecanismos de bajo nivel necesarios para implementar procesos y las políticas de nivel superior necesarias para programarlos de manera inteligente. Combinando mecanismos y políticas, ampliaremos nuestra comprensión de cómo un sistema operativo virtualiza la CPU.

---

<sup>1</sup>Sí, el estado zombie. Al igual que los zombis reales, estos zombis son relativamente fáciles de matar. Sin embargo, normalmente se recomiendan técnicas diferentes.

## APARTE: TÉRMINOS CLAVE DEL PROCESO

- El proceso es la principal abstracción del sistema operativo de un programa en ejecución. En cualquier momento, el proceso puede describirse por su estado: el contenido de la memoria en su espacio de direcciones, el contenido de los registros de la CPU (incluido el contador de programa y el puntero de pila, entre otros), e información sobre E/S (como archivos abiertos que se pueden leer o escrito).
- La API de procesos consta de llamadas que los programas pueden realizar relacionadas con procesos. Normalmente, esto incluye creación, destrucción y otras llamadas útiles.
- Los procesos existen en uno de los muchos estados de proceso diferentes, incluidos en ejecución, listo para ejecutarse y bloqueado. Diferentes eventos (por ejemplo, conseguir programado o desprogramado, o esperando a que se complete una E/S) realizar la transición de un proceso de uno de estos estados al otro.
- Una lista de procesos contiene información sobre todos los procesos del sistema. Cada entrada se encuentra en lo que a veces se llama proceso. bloque de control (PCB), que en realidad es solo una estructura que contiene información sobre un proceso específico.

## Referencias

[BH70] "El núcleo de un sistema de multiprogramación" de Per Brinch Hansen. Communications of the ACM, Volumen 13:4, abril de 1970. Este artículo presenta uno de los primeros microkernels en la historia de los sistemas operativos, llamado Nucleus. La idea de sistemas más pequeños y minimalistas es un tema que surge repetidamente en la historia de los sistemas operativos; Todo comenzó con el trabajo de Brinch Hansen que aquí se describe.

[CK+08] "El sistema operativo xv6" por Russ Cox, Frans Kaashoek, Robert Morris, Nickolai Zeldovich. De: <https://github.com/mit-pdos/xv6-public>. El sistema operativo pequeño y real más genial del mundo. Descárgalo y juega con él para aprender más sobre los detalles de cómo funcionan realmente los sistemas operativos. Hemos estado utilizando una versión anterior (2012-01-30-1-gf1c41342) y, por lo tanto, es posible que algunos ejemplos del libro no coincidan con la última versión de la fuente.

[DV66] "Programación de semántica para computaciones multiprogramadas" por Jack B. Dennis, Earl C. Van Horn. Comunicaciones de la ACM, Tomo 9, Número 3, marzo de 1966. Este artículo definió muchos de los primeros términos y conceptos relacionados con la construcción de sistemas multiprogramados.

[L+75] "Separación de políticas y mecanismos en Hydra" por R. Levin, E. Cohen, W. Corwin, F. Pollack, W. Wulf. SOSP '75, Austin, Texas, noviembre de 1975. Uno de los primeros artículos sobre cómo estructurar sistemas operativos en un sistema operativo de investigación conocido como Hydra. Si bien Hydra nunca se convirtió en un sistema operativo convencional, algunas de sus ideas influyeron en los diseñadores de sistemas operativos.

[V+65] "Estructura del Supervisor Multics" por VA Vyssotsky, FJ Corbato, RM Graham.

Fall Joint Computer Conference, 1965. Uno de los primeros artículos sobre Multics, que describía muchas de las ideas y términos básicos que encontramos en los sistemas modernos. Parte de la visión detrás de la informática como utilidad finalmente se está haciendo realidad en los sistemas de nube modernos.

## Tarea (Simulación)

Este programa, Process-run.py, le permite ver cómo cambian los estados de los procesos a medida que los programas se ejecutan y usan la CPU (por ejemplo, realizar una instrucción de adición) o realizan E/S (por ejemplo, enviar una solicitud a un disco y esperar que se complete). Consulte el archivo LÉAME para obtener más detalles.

Preguntas 1.

Ejecute Process-run.py con los siguientes indicadores: -l 5:100,5:100.

¿Cuál debería ser la utilización de la CPU (por ejemplo, el porcentaje de tiempo que la CPU está en uso)? ¿Por qué sabes esto? Utilice los indicadores -c y -p para ver si tenía razón.

2. Ahora ejecute con estas opciones: ./process-run.py -l 4:100,1:0.

Estos indicadores especifican un proceso con 4 instrucciones (todas para usar la CPU) y uno que simplemente emite una E/S y espera a que se complete.

¿Cuánto tiempo lleva completar ambos procesos? Utilice -c y -p para averiguar si tenía razón.

3. Cambie el orden de los procesos: -l 1:0,4:100. ¿Qué pasa ahora? ¿Importa cambiar el orden? ¿Por qué? (Como siempre, usa -c y -p para ver si tenías razón)

4. Ahora exploraremos algunas de las otras banderas. Un indicador importante es -S, que determina cómo reacciona el sistema cuando un proceso emite una E/S. Con el indicador configurado en ENCENDIDO FINAL, el sistema NO cambiará a otro proceso mientras uno esté realizando E/S, sino que esperará hasta que el proceso finalice por completo. ¿Qué sucede cuando ejecuta los siguientes dos procesos (-l 1:0,4:100 -c -S SWITCH ON END), uno haciendo E/S y el otro haciendo trabajo de CPU?

- - -

5. Ahora, ejecute los mismos procesos, pero con el comportamiento de conmutación configurado para cambiar a otro proceso cada vez que uno esté ESPERANDO E/S (-l 1:0,4:100 -c -S SWITCH ON IO). ¿Qué pasa ahora? Utilice -c y -p para confirmar que tiene razón.

6. Otro comportamiento importante es qué hacer cuando se completa una E/S. Con -l IO RUN LATER, cuando se completa una E/S, el proceso que la emitió no necesariamente se ejecuta de inmediato; más bien, todo lo que estaba funcionando en ese momento sigue funcionando. ¿Qué sucede cuando ejecutas esta combinación de procesos? (Ejecutar ./process-run.py -l 3:0,5:100,5:100,5:100 -S ENCENDER IO -l IO EJECUTAR MÁS TARDE -c -p) ¿Se están utilizando eficazmente los recursos del sistema?

- - - - -

7. Ahora ejecute los mismos procesos, pero con -l IO RUN IMMEDIATE configurado, que ejecuta inmediatamente el proceso que emitió la E/S. ¿En qué se diferencia este comportamiento? ¿Por qué podría ser una buena idea ejecutar un proceso que acaba de completar una E/S nuevamente?

8. Ahora ejecute algunos procesos generados aleatoriamente: -s 1 -l 3:50,3:50 o -s 2 -l 3:50,3:50 o -s 3 -l 3:50,3:50. Vea si puede predecir cómo resultará el rastro. ¿Qué sucede cuando usas el indicador -I IO RUN INMEDIATO versus -I IO RUN DESPUÉS? ¿Qué sucede cuando usas -S SWITCH ON IO versus -S SWITCH ON END? - - - -

## Interludio: API de proceso

### APARTE: INTERLUDES

Los interludios cubrirán aspectos más prácticos de los sistemas, incluido un enfoque particular en las API del sistema operativo y cómo usarlas. Si no te gustan las cosas prácticas, puedes saltarte estos interludios. Pero te deberían gustar las cosas prácticas, porque, bueno, generalmente son útiles en la vida real; Las empresas, por ejemplo, no suelen contratarte por tus habilidades no prácticas.

En este interludio, analizamos la creación de procesos en sistemas UNIX . UNIX presenta una de las formas más intrigantes de crear un nuevo proceso con un par de llamadas al sistema: fork() y exec(). Una tercera rutina, esperar(), puede ser utilizada por un proceso que desee esperar a que se complete un proceso que ha creado. Ahora presentamos estas interfaces con más detalle, con algunos ejemplos sencillos para motivarnos. Y así, nuestro problema:

### CRUX : CÓMO CREAR Y CONTROLAR PROCESOS

¿Qué interfaces debería presentar el sistema operativo para la creación y el control de procesos? ¿Cómo deberían diseñarse estas interfaces para permitir una funcionalidad poderosa, facilidad de uso y alto rendimiento?

### 5.1 La llamada al sistema fork()

La llamada al sistema fork() se utiliza para crear un nuevo proceso [C63]. Sin embargo, tenga cuidado: sin duda es la rutina más extraña que jamás haya conocido<sup>1</sup> . Más específicamente, tiene un programa en ejecución cuyo código se parece al que ve en la Figura 5.1; examine el código, o mejor aún, escríbalo y ejecútelo usted mismo.

---

<sup>1</sup>Bueno, vale, admitimos que no lo sabemos con seguridad; ¿Quién sabe qué rutinas llamas cuando nadie te mira? Pero fork() es bastante extraño, sin importar cuán inusuales sean sus patrones de llamadas a rutinas.

```

1 #incluir <stdio.h>
2 #incluir <stdlib.h>
3 #incluir <unistd.h>
4
5 int principal(int argc, char *argv[]) {
6     printf("hola mundo (pid:%d)\n", (int) getpid());
7     int rc = bifurcación();
8     si (rc < 0) {
9         // la bifurcación falló
10        fprintf(stderr, "error en la bifurcación\n");
11        salir(1);
12    } si no (rc == 0) {
13        // hijo (nuevo proceso)
14        printf("hola, soy niño (pid:%d)\n", (int) getpid());
15    } demás {
16        // el padre sigue este camino (principal)
17        printf("hola, soy padre de %d (pid:%d)\n",
18               rc, (int) getpid());
19    }
20    devolver 0;
21 }
22 }
```

Figura 5.1: Llamar a fork() (p1.c)

Cuando ejecute este programa (llamado p1.c), verá lo siguiente:

```

mensaje> ./p1
hola mundo (pid: 29146)
hola, soy padre de 29147 (pid:29146)
hola soy un niño (pid:29147)
mensaje>
```

Entendamos lo que sucedió con más detalle en p1.c. cuando  
Cuando comienza a ejecutarse por primera vez, el proceso imprime un mensaje de hola  
mundo; En ese mensaje se incluye su identificador de proceso, también conocido como PID. El  
el proceso tiene un PID de 29146; En sistemas UNIX , el PID se utiliza para nombrar  
el proceso si uno quiere hacer algo con el proceso, como (por ejemplo)  
ejemplo) detener su ejecución. Hasta ahora, todo bien.

Ahora comienza la parte interesante. El proceso llama al sistema fork()  
llamada, que el sistema operativo proporciona como una forma de crear un nuevo proceso. lo extraño  
parte: el proceso que se crea es una copia (casi) exacta del proceso de llamada.  
Eso significa que para el sistema operativo ahora parece que hay dos copias de  
el programa p1 se está ejecutando y ambos están a punto de regresar de la bifurcación()  
llamada al sistema. El proceso recién creado (llamado niño, en contraste con el  
creando parent) no comienza a ejecutarse en main(), como se podría esperar  
(tenga en cuenta que el mensaje "hola, mundo" solo se imprimió una vez); más bien,  
simplemente cobra vida como si se hubiera llamado a fork() .

```

1 #incluir <stdio.h>
2 #incluir <stdlib.h>
3 #incluir <unistd.h>
4 #incluir <sys/wait.h>
5
6 int principal(int argc, char *argv[]) {
7     printf("hola mundo (pid:%d)\n", (int) getpid());
8     int rc = bifurcación();
9     si (rc < 0) {                                // la bifurcación falló; salida
10        fprintf(stderr, "error en la bifurcación\n");
11        salir(1);
12    } else if (rc == 0) { // hijo (nuevo proceso)
13        printf("hola, soy niño (pid:%d)\n", (int) getpid());
14    } más { int                                // el padre sigue este camino (principal)
15        rc_wait = esperar(NULL);
16        printf("hola, soy padre de %d (rc_wait:%d) (pid:%d)\n",
17               rc, rc_wait, (int) getpid());
18    }
19    devolver 0;
20}
21

```

Figura 5.2: Llamando a fork() y wait() (p2.c)

Te habrás dado cuenta: el niño no es una copia exacta. Específicamente, aunque ahora tiene su propia copia del espacio de direcciones (es decir, su propia copia privada memoria), sus propios registros, su propia PC, etc., el valor que devuelve para la persona que llama a fork() es diferente. Específicamente, mientras el padre recibe el PID del niño recién creado, el niño recibe un código de retorno de cero. Esta diferenciación es útil, porque entonces es sencillo escribir el código que maneja los dos casos diferentes (como arriba).

También habrás notado: el resultado (de p1.c) no es determinista. Cuando se crea el proceso hijo, ahora hay dos procesos activos en el sistema que nos importa: el padre y el niño. Suponiendo que nosotros se ejecutan en un sistema con una sola CPU (para simplificar), entonces el niño o el padre podrían correr en ese punto. En nuestro ejemplo (arriba), el padre lo hizo y, por lo tanto, imprimió su mensaje primero. En otros casos, el Podría suceder lo contrario, como mostramos en este seguimiento de salida:

```

mensaje> ./p1
hola mundo (pid: 29146)
hola soy un niño (pid:29147)
hola, soy padre de 29147 (pid:29146)
mensaje>

```

El programador de CPU, un tema que discutiremos con gran detalle pronto, determina qué proceso se ejecuta en un momento dado; Debido a que el programador es complejo, generalmente no podemos hacer suposiciones sólidas sobre lo que

elegirá hacer y, por tanto, qué proceso se ejecutará primero. Resulta que este no determinismo conduce a algunos problemas interesantes, particularmente en programas multiproceso; por lo tanto, veremos mucho más no determinismo cuando estudiemos la concurrencia en la segunda parte del libro.

## 5.2 La llamada al sistema wait()

Hasta ahora, no hemos hecho mucho: simplemente creamos un niño que imprime un mensaje y sale. A veces, resulta que es bastante útil para un padre esperar a que un proceso hijo termine lo que ha estado haciendo. Esta tarea se logra con la llamada al sistema `wait()` (o su hermano más completo `waitpid()`); consulte la Figura 5.2 para obtener más detalles.

En este ejemplo (p2.c), el proceso padre llama a `wait()` para retrasar su ejecución hasta que el hijo termine de ejecutarse. Cuando el niño termina, `wait()` regresa al padre.

Agregar una llamada `wait()` al código anterior hace que la salida sea determinista. ¿Puedes ver por qué? Adelante, piénsalo.

(esperando que pienses....y listo)

Ahora que has pensado un poco, aquí está el resultado:

```
mensaje> ./p2 hola
mundo (pid:29266) hola, soy niño
(pid:29267) hola, soy padre de 29267
(rc_wait:29267) (pid:29266) mensaje>
```

Con este código, ahora sabemos que el niño siempre imprimirá primero. ¿Por qué sabemos eso? Bueno, podría simplemente ejecutarse primero, como antes, y así imprimirse antes que el padre. Sin embargo, si el padre se ejecuta primero, llamará inmediatamente a `wait()`; esta llamada al sistema no regresará hasta que el niño haya corrido y salido<sup>2</sup>. Por lo tanto, incluso cuando el padre se ejecuta primero, espera cortésmente a que el hijo termine de ejecutarse, luego espera () regresa y luego el padre imprime su mensaje.

## 5.3 Finalmente, la llamada al sistema exec()

Una pieza final e importante de la API de creación de procesos es la llamada al sistema `exec()`<sup>3</sup>. Esta llamada al sistema es útil cuando desea ejecutar un programa que es diferente del programa que llama. Por ejemplo, llamando a `fork()`

---

<sup>2</sup>Hay algunos casos en los que `wait()` regresa antes de que el niño salga; lea la página de manual para más detalles, como siempre. Y tenga cuidado con las declaraciones absolutas e incondicionales que hace este libro, como "el niño siempre imprimirá primero" o "UNIX es lo mejor del mundo, incluso mejor que el helado".

<sup>3</sup>En Linux, hay seis variantes de `exec()`: `exec`, `execp()`, `execle()`, `execv()`, `execvp()` y `execvpe()`. Lea las páginas de manual para obtener más información.

```

1 #incluir <stdio.h>
2 #incluir <stdlib.h>
3 #incluir <unistd.h>
4 #incluir <cadena.h>
5 #incluir <sys/wait.h>
6
7 int principal(int argc, char *argv[]) {
8     printf("hola mundo (pid:%d)\n", (int) getpid());
9     int rc = bifurcación();
10    si (rc < 0) {                                // la bifurcación falló; salida
11        fprintf(stderr, "error en la bifurcación\n");
12        salir(1);
13    } else if (rc == 0) { // hijo (nuevo proceso)
14        printf("hola, soy niño (pid:%d)\n", (int) getpid());
15        char *myargs[3];
16        myargs[0] = strdup("wc"); // programa: "wc" (recuento de palabras)
17        myargs[1] = strdup("p3.c"); // argumento: archivo a contar
18        myargs[2] = NULL;           // marca el final de la matriz
19        execvp(myargs[0], myargs); // ejecuta el recuento de palabras
20        printf("esto no debería imprimirse");
21    } más { int                         // el padre sigue este camino (principal)
22        rc_wait = esperar(NULL);
23        printf("hola, soy padre de %d (rc_wait:%d) (pid:%d)\n",
24               rc, rc_wait, (int) getpid());
25    }
26    devolver 0;
27 }
28

```

Figura 5.3: Llamar a fork(), wait() y exec() (p3.c)

en p2.c sólo es útil si desea seguir ejecutando copias del mismo programa. Sin embargo, a menudo desea ejecutar un programa diferente; ejecutivo() hace precisamente eso (Figura 5.3).

En este ejemplo, el proceso hijo llama a execvp() para ejecutar el programa wc, que es el programa de conteo de palabras. De hecho, funciona con WC . el archivo fuente p3.c, indicándonos así cuántas líneas, palabras y bytes hay encontrado en el archivo:

```

mensaje> ./p3
hola mundo (pid: 29383)
hola soy un niño (pid:29384)
      29 107 1030 p3.c
hola, soy padre de 29384 (rc_wait:29384) (pid:29383)
mensaje>

```

La llamada al sistema fork() es extraña; su cómplice, exec(), no es tan normal tampoco. Qué hace: dado el nombre de un ejecutable (por ejemplo, wc), y algunos argumentos (por ejemplo, p3.c), carga código (y datos estáticos) desde ese

**CONSEJO: HACERLO BIEN ( LEY DE LAMPSON )**

Como afirma Lampson en su prestigioso "Consejos para el diseño de sistemas informáticos" [183], "Hágalo bien. Ni la abstracción ni la simplicidad sustituyen a hacerlo bien". A veces, sólo tienes que hacer lo correcto y, cuando lo haces, es mucho mejor que las alternativas. Hay muchas formas de diseñar API para la creación de procesos; sin embargo, la combinación de fork() y exec() es simple e inmensamente poderosa. En este caso, los diseñadores de UNIX simplemente lo hicieron bien. Y como Lampson muchas veces "lo hizo bien", nombramos la ley en su honor.

ejecutable y sobrescribe su segmento de código actual (y datos estáticos actuales) con él; el montón, la pila y otras partes del espacio de memoria del programa se reinicializan. Luego, el sistema operativo simplemente ejecuta ese programa y pasa cualquier argumento como argumento de ese proceso. Por tanto, no crea un nuevo proceso; más bien, transforma el programa que se está ejecutando actualmente (anteriormente p3) en un programa en ejecución diferente (wc). Después de exec() en el niño, es casi como si p3.c nunca se hubiera ejecutado; una llamada exitosa a exec() nunca regresa.

## 5.4 ¿Por qué? Motivando a la API

Por supuesto, es posible que tenga una gran pregunta: ¿por qué construiríamos una interfaz tan extraña para lo que debería ser el simple acto de crear un nuevo proceso? Bueno, resulta que la separación de fork() y exec() es esencial en la construcción de un shell UNIX , porque permite que el shell ejecute código después de la llamada a fork() pero antes de la llamada a exec(); este código puede alterar el entorno del programa que está a punto de ejecutarse y, por lo tanto, permite crear fácilmente una variedad de características interesantes.

El shell es sólo un programa de usuario<sup>4</sup> . Le muestra un mensaje y luego espera a que escriba algo en él. Luego escribe un comando (es decir, el nombre de un programa ejecutable, más cualquier argumento) en él; en la mayoría de los casos, el shell descubre en qué parte del sistema de archivos reside el ejecutable, llama a fork() para crear un nuevo proceso hijo para ejecutar el comando, llama a alguna variante de exec() para ejecutar el comando y luego espera a que comando para completar llamando a wait(). Cuando el niño completa, el shell regresa de wait() e imprime un mensaje nuevamente, listo para su siguiente comando.

La separación de fork() y exec() permite que el shell haga un montón de cosas útiles con bastante facilidad. Por ejemplo:

```
símbolo> wc p3.c > nuevoarchivo.txt
```

---

<sup>4</sup>Y hay muchas conchas; tcsh, bash y zsh , por nombrar algunos. Deberías elegir uno, leer sus páginas de manual y aprender más sobre él; todos los expertos en UNIX lo hacen.

En el ejemplo anterior, la salida del programa wc se redirige a el archivo de salida newfile.txt (el signo mayor que es como se indica dicha redirección). La forma en que el shell realiza esta tarea es bastante simple: cuando se crea el hijo, antes de llamar a exec(), el shell se cierra. salida estándar y abre el archivo newfile.txt. Al hacerlo, cualquier salida del programa wc que se ejecutará próximamente se envía al archivo de la pantalla.

La Figura 5.4 (página 8) muestra un programa que hace exactamente esto. La razón Esta redirección funciona se debe a una suposición sobre cómo funciona el sistema operativo. El sistema gestiona los descriptores de archivos. Específicamente, los sistemas UNIX comienzan a buscar para descriptores de archivos gratuitos en cero. En este caso, STDOUT\_FILENO será el el primero disponible y así ser asignado cuando se llama a open(). Escrituras posteriores del proceso hijo en el descriptor de archivo de salida estándar, por ejemplo mediante rutinas como printf(), se enrutarán de forma transparente al archivo recién abierto en lugar de a la pantalla.

Aquí está el resultado de ejecutar el programa p4.c :

```
mensaje> ./p4
mensaje> gato p4.salida
      32      109 846 p4.c
mensaje>
```

Notarás (al menos) dos datos interesantes sobre este resultado. Primero, cuando se ejecuta p4 , parece como si no hubiera pasado nada; el caparazón simplemente se imprime el símbolo del sistema y está inmediatamente listo para su siguiente comando. Sin embargo, ese no es el caso; el programa p4 efectivamente llamó a fork() para cree un nuevo hijo y luego ejecute el programa wc mediante una llamada a execvp(). No ve ningún resultado impreso en la pantalla porque ha sido redirigido al archivo p4.output. En segundo lugar, puedes ver que cuando tomamos el archivo de salida, se encuentra todo el resultado esperado al ejecutar wc . Genial, ¿verdad?

Las tuberías UNIX se implementan de manera similar, pero con la función pipe() llamada al sistema. En este caso, la salida de un proceso está conectada a una tubería interna (es decir, una cola) y la entrada de otro proceso está conectada a esa misma tubería; por lo tanto, el resultado de un proceso se utiliza sin problemas como entrada al siguiente, y se pueden encadenar largas y útiles cadenas de comandos juntos. Como ejemplo sencillo, considere buscar una palabra en un archivo y luego contar cuántas veces aparece dicha palabra; con las tuberías y los servicios públicos, grep y wc, es fácil; simplemente escriba grep -o foo archivo | baño -l en el símbolo del sistema y maravíllese con el resultado.

Finalmente, si bien acabamos de esbozar la API del proceso en un nivel alto, Hay muchos más detalles sobre estas llamadas que aprender y asimilado; Aprenderemos más, por ejemplo, sobre descriptores de archivos cuando Hablaremos sobre sistemas de archivos en la tercera parte del libro. Por ahora basta decir que la combinación fork()/exec() es una forma poderosa de crear y manipular procesos.

```

1 #incluir <stdio.h>
2 #incluir <stdlib.h>
3 #incluir <unistd.h>
4 #incluir <cadena.h>
5 #incluir <fcntl.h>
6 #incluir <sys/wait.h>
7
8 int principal(int argc, char *argv[]) {
9     int rc = bifurcación();
10    si (rc < 0) {
11        // la bifurcación falló
12        fprintf(stderr, "error en la bifurcación\n");
13        salir(1);
14    } si no (rc == 0) {
15        // hijo: redirigir la salida estándar a un archivo
16        cerrar(STDOUT_FILENO);
17        open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
18
19        // ahora ejecuta "wc"...
20        char *myargs[3];
21        myargs[0] = strdup("wc"); // programa: wc (recuento de palabras)
22        myargs[1] = strdup("p4.c"); // arg: archivo para contar
23        myargs[2] = NULL;           // marca el final de la matriz
24        execvp(myargs[0], myargs); // ejecuta el recuento de palabras
25    } demás {
26        // el padre sigue este camino (principal)
27        int rc_wait = esperar(NULL);
28    }
29    devolver 0;
30 }

```

Figura 5.4: Todo lo anterior con redirección (p4.c)

## 5.5 Control de procesos y usuarios

Más allá de fork(), exec() y wait(), existen muchas otras interfaces para interactuar con procesos en sistemas UNIX . Por ejemplo, el

La llamada al sistema kill() se utiliza para enviar señales a un proceso, incluidas directivas para pausar, morir y otros imperativos útiles. Por conveniencia,

En la mayoría de los shells UNIX , ciertas combinaciones de teclas están configuradas para entregar una señal específica al proceso actualmente en ejecución; Por ejemplo, control-c envía un SIGINT (interrupción) al proceso (normalmente terminando

it) y control-z envía una señal SIGTSTP (detener) pausando así el proceso

en mitad de la ejecución (puede reanudarla más tarde con un comando, por ejemplo, el fg comando incorporado que se encuentra en muchos shells).

Todo el subsistema de señales proporciona una rica infraestructura para entregar eventos externos a los procesos, incluidas las formas de recibirlas y procesarlos.

señales dentro de procesos individuales y formas de enviar señales a procesos individuales, así como a grupos de procesos completos. Para utilizar esta forma de comunicación

**APARTE: RTFM — LEA LAS PÁGINAS MAN**

Muchas veces en este libro, cuando nos referimos a una llamada al sistema o a una biblioteca en particular, le indicaremos que lea las páginas del manual o , para abreviar, las páginas del manual . Las páginas de manual son la forma original de documentación que existe en los sistemas UNIX ; darse cuenta de que fueron creados antes de que existiera la cosa llamada web .

Dedicar algún tiempo a leer páginas de manual es un paso clave en el crecimiento de un programador de sistemas; Hay toneladas de datos útiles escondidos en esas páginas. Algunas páginas particularmente útiles para leer son las páginas de manual para cualquier shell que esté utilizando (por ejemplo, tcsh o bash), y ciertamente para cualquier llamada al sistema que realice su programa (para ver qué valores de retorno y condiciones de error existen).

Finalmente, leer las páginas de manual puede ahorrarle cierta vergüenza. Cuando les pregunta a sus colegas sobre algunas complejidades de fork(), es posible que simplemente respondan: "RTFM". Esta es la forma en que sus colegas le instan amablemente a leer las páginas man. La F en RTFM sólo añade un poco de color a la frase...

comunicación, un proceso debe utilizar la llamada al sistema signal() para "captar" varias señales; hacerlo garantiza que cuando se entrega una señal particular a un proceso, éste suspenderá su ejecución normal y ejecutará un fragmento de código particular en respuesta a la señal. Lea en otro lugar [SR05] para aprender más sobre las señales y sus muchas complejidades.

Naturalmente, esto plantea la pregunta: ¿quién puede enviar una señal a un proceso y quién no? Generalmente, los sistemas que utilizamos pueden tener varias personas usándolos al mismo tiempo; Si una de estas personas puede enviar arbitrariamente señales como SIGINT (para interrumpir un proceso y probablemente terminarlo), la usabilidad y seguridad del sistema se verán comprometidas. Como resultado, los sistemas modernos incluyen una fuerte concepción de la noción de usuario. El usuario, después de ingresar una contraseña para establecer credenciales, inicia sesión para obtener acceso a los recursos del sistema. A continuación, el usuario puede iniciar uno o varios procesos y ejercer un control total sobre ellos (pausarlos, finalizarlos, etc.). Los usuarios generalmente sólo pueden controlar sus propios procesos; Es trabajo del sistema operativo distribuir recursos (como CPU, memoria y disco) a cada usuario (y sus procesos) para cumplir con los objetivos generales del sistema.

## 5.6 Herramientas útiles

Hay muchas herramientas de línea de comandos que también son útiles. Por ejemplo, usar el comando ps le permite ver qué procesos se están ejecutando; lea las páginas de manual para conocer algunos indicadores útiles para pasar a ps. La parte superior de herramientas también es bastante útil, ya que muestra los procesos del sistema y cuánta CPU y otros recursos están consumiendo. Con humor, muchas veces, cuando lo ejecutas, top afirma que es el principal consumidor de recursos; tal vez sea un poco ególatra. El comando kill se puede utilizar para enviar mensajes arbitrarios.

**APARTE: EL SUPERUSUARIO (ROOT)**

Un sistema generalmente necesita un usuario que pueda administrarlo y no esté limitado como lo están la mayoría de los usuarios. Dicho usuario debería poder finalizar un proceso arbitrario (por ejemplo, si está abusando del sistema de alguna manera), incluso aunque ese proceso no haya sido iniciado por este usuario. Un usuario así también debería poder ejecutar comandos potentes como apagar (que, como era de esperar, apaga el sistema). En los sistemas basados en UNIX, estas capacidades especiales se otorgan al superusuario (a veces llamado root). Si bien la mayoría de los usuarios no pueden eliminar los procesos de otros usuarios, el superusuario sí puede. Ser raíz es muy parecido a ser Spider-Man: un gran poder conlleva una gran responsabilidad [QI15]. Por lo tanto, para aumentar la seguridad (y evitar errores costosos), normalmente es mejor ser un usuario habitual; Si necesitas ser root, anda con cuidado, ya que todos los poderes destructivos del mundo de la informática están ahora a tu alcance.

señales a los procesos, al igual que el killall, un poco más fácil de usar . Asegúrese de utilizarlos con cuidado; Si accidentalmente matas tu administrador de ventanas, la computadora frente a la cual estás sentado puede volverse bastante difícil de usar.

Finalmente, hay muchos tipos diferentes de medidores de CPU que puede utilizar para obtener una idea rápida de la carga de su sistema; por ejemplo, siempre mantenemos MenuMeters (del software Raging Menace) ejecutándose en nuestras barras de herramientas de Macintosh, para que podamos ver cuánta CPU se está utilizando en cualquier momento. En general, cuanta más información sobre lo que está pasando, mejor.

## 5.7 Resumen

Hemos introducido algunas de las API relacionadas con la creación de procesos UNIX : fork(), exec() y wait(). Sin embargo, sólo hemos rozado la superficie. Para más detalles, lea Stevens y Rago [SR05], por supuesto, particularmente los capítulos sobre Control de Procesos, Relaciones de Procesos y Señales; hay mucho que extraer de la sabiduría que contiene.

Si bien nuestra pasión por la API del proceso UNIX sigue siendo fuerte, también debemos señalar que esa positividad no es uniforme. Por ejemplo, un artículo reciente de investigadores de sistemas de Microsoft, la Universidad de Boston y ETH en Suiza detalla algunos problemas con fork() y aboga por otras API de creación de procesos más simples, como spawn() [B+19]. Léelo y el trabajo relacionado al que se refiere para comprender este punto de vista diferente. Si bien en general es bueno confiar en este libro, recuerde también que los autores tienen opiniones; Es posible que esas opiniones no (siempre) sean tan compartidas como podría pensar.

## APARTE: TÉRMINOS CLAVE DE LA API DE PROCESO

- Cada proceso tiene un nombre; en la mayoría de los sistemas, ese nombre es un número conocido como ID de proceso (PID).
- La llamada al sistema fork() se utiliza en sistemas UNIX para crear un nuevo proceso. El creador se llama padre; el proceso recién creado se llama hijo. Como ocurre a veces en la vida real [J16], el proceso hijo es una copia casi idéntica del padre.
- La llamada al sistema wait() permite a un parent esperar a que su hijo complete la ejecución.
- La familia de llamadas al sistema exec() permite a un niño liberarse de su similitud con su parent y ejecutar un programa completamente nuevo.
- Un shell UNIX comúnmente usa fork(), wait() y exec() para ejecutar comandos de usuario; la separación de fork y exec permite funciones como redirección de entrada/salida, canalizaciones y otras funciones interesantes, todo sin cambiar nada en los programas que se ejecutan. • El control de procesos está disponible en forma de señales, lo que puede causar trabajos para detener, continuar o incluso terminar.
- Qué procesos pueden ser controlados por una persona en particular está encapsulado en la noción de usuario; el sistema operativo permite que varios usuarios accedan al sistema y garantiza que los usuarios solo puedan controlar sus propios procesos.
- Un superusuario puede controlar todos los procesos (y de hecho hacer muchas otras cosas); este papel debe asumirse con poca frecuencia y con precaución por razones de seguridad.

## Referencias

[B+19] "Una bifurcación() en el camino" de Andrew Baumann, Jonathan Appavoo, Orran Krieger, Tim-othy Roscoe. HotOS '19, Bertinoro, Italia. Un artículo divertido lleno de rabia por el tenedor. Léalo para obtener un punto de vista opuesto sobre la API del proceso UNIX . Presentado en el siempre animado taller HotOS, donde los investigadores de sistemas presentan opiniones extremas con la esperanza de impulsar a la comunidad en nuevas direcciones.

[C63] "Diseño de un sistema multiprocesador" por Melvin E. Conway. AFIPS '63 Fall Joint Computer Conference , Nueva York, EE.UU. 1963. Uno de los primeros artículos sobre cómo diseñar sistemas multiprocesamiento; puede ser el primer lugar donde se utilizó el término fork() en la discusión sobre la generación de nuevos procesos.

[DV66] "Programación de semántica para computaciones multiprogramadas" por Jack B. Dennis y Earl C. Van Horn. Communications of the ACM, Volumen 9, Número 3, marzo de 1966. Un artículo clásico que describe los conceptos básicos de los sistemas informáticos multiprogramados. Sin duda tuvo una gran influencia en Project MAC, Multics y, finalmente, UNIX.

[J16] "¡Podrían ser gemelos!" por Phoebe Jackson-Edwards. El correo diario. 1 de marzo de 2016. Esta pieza periodística contundente muestra un montón de fotografías de padres e hijos extrañamente similares y, francamente, es algo fascinante. Adelante, pierde dos minutos de tu vida y compruébalo. ¡Pero no olvides volver aquí! Éste, en un microcosmos, es el peligro de navegar por la web.

[L83] "Consejos para el diseño de sistemas informáticos" de Butler Lampson. ACM Operating Systems Review, volumen 15:5, octubre de 1983. Los famosos consejos de Lampson sobre cómo diseñar sistemas informáticos. Deberías leerlo en algún momento de tu vida, y probablemente en muchos momentos de tu vida.

[QI15] "Un gran poder conlleva una gran responsabilidad" por The Quote Investigator. Disponible: <https://quoteinvestigator.com/2015/07/23/great-power>. El investigador de citas concluye que la primera mención de este concepto es de 1793, en una colección de decretos emitidos en la Convención Nacional Francesa. La cita específica: "Ils doivent envisager qu'une grande responsabilité est la suite inséparable d'un grand pouvoir", que se traduce aproximadamente como "Deben considerar que una gran responsabilidad surge inseparablemente de un gran poder". Recién en 1962 aparecieron en Spider-Man las siguientes palabras: "...con un gran poder también debe venir: ¡una gran responsabilidad!" Entonces parece que la Revolución Francesa recibe el crédito por esto, no Stan Lee. Lo siento, Stan.

[SR05] "Programación avanzada en el entorno UNIX " por W. Richard Stevens, Stephen A. Rago. Addison-Wesley, 2005. Todos los matices y sutilezas del uso de las API de UNIX se encuentran aquí. ¡Compra este libro! ¡Léelo! Y lo más importante, vívelo.

## Tarea (Simulación)

Esta tarea de simulación se centra en fork.py, un sencillo simulador de creación de procesos que muestra cómo se relacionan los procesos en un único árbol "familiar". Lea el archivo README correspondiente para obtener detalles sobre cómo ejecutar el simulador.

### Preguntas 1.

- Ejecute `./fork.py -s 10` y vea qué acciones se toman. ¿Puedes predecir cómo se verá el árbol de procesos en cada paso? Utilice la opción `-c` para comprobar sus respuestas. Pruebe algunas semillas aleatorias diferentes (`-s`) o agregue más acciones (`-a`) para dominarlo.
2. Un control que le brinda el simulador es el porcentaje de bifurcación, controlado por el indicador `-f`. Cuanto más alto sea, más probable es que la siguiente acción sea una bifurcación; cuanto más bajo sea, más probable es que la acción sea una salida. Ejecute el simulador con una gran cantidad de acciones (por ejemplo, `-a 100`) y varíe el porcentaje de bifurcación de 0,1 a 0,9. ¿Cómo crees que se verán los árboles de proceso final resultantes a medida que cambie el porcentaje? Comprueba tu respuesta con `-c`.
  3. Ahora, cambie la salida usando el indicador `-t` (por ejemplo, ejecute `./fork.py -t`). Dado un conjunto de árboles de procesos, ¿puedes decir qué acciones se tomaron?
  4. Una cosa interesante a tener en cuenta es lo que sucede cuando un niño sale; ¿Qué sucede con sus hijos en el árbol de procesos? Para estudiar esto, creamos un ejemplo específico: `./fork.py -A a+b,b+c,c+d,c+e,c-`. En este ejemplo, el proceso 'a' crea 'b', que a su vez crea 'c', que luego crea 'd' y 'e'. Sin embargo, entonces, 'c' sale. ¿Cómo crees que le debería gustar al árbol de procesos después de la salida? ¿Qué pasa si usas la bandera `-R`? Obtenga más información sobre lo que sucede con los procesos huérfanos por su cuenta para agregar más contexto.
  5. Una última bandera a explorar es la bandera `-F`, que omite los pasos intermedios y solo solicita completar el árbol de procesos final. Ejecute `./fork.py -F` y vea si puede escribir el árbol final observando la serie de acciones generadas. Utilice diferentes semillas al azar para intentar esto varias veces.
  6. Finalmente, use `-t` y `-F` juntos. Esto muestra el árbol de procesos final, pero luego le pide que complete las acciones que tuvieron lugar. Al mirar el árbol, ¿puedes determinar las acciones exactas que tuvieron lugar?  
¿En qué casos puedes saberlo? ¿En cuál no lo sabes? Pruebe algunas semillas aleatorias diferentes para profundizar en esta pregunta.

**APARTE: TAREAS DE CODIFICACIÓN**

Las tareas de codificación son pequeños ejercicios en los que se escribe código para ejecutarlo en una máquina real y adquirir algo de experiencia con algunas API básicas del sistema operativo. Después de todo, usted es (probablemente) un informático y, por lo tanto, le gustaría codificar, ¿verdad? Si no lo hace, siempre existe la teoría de la informática, pero eso es bastante difícil. Por supuesto, para convertirse verdaderamente en un experto, debe dedicar más de un poco de tiempo a manipular la máquina; de hecho, encuentre todas las excusas que pueda para escribir código y ver cómo funciona. Dedica tiempo y conviértete en el maestro sabio que sabes que puedes ser.

**Tarea (Código)**

En esta tarea, deberá familiarizarse con las API de gestión de procesos sobre las que acaba de leer. No te preocupes: ¡es incluso más divertido de lo que parece! En general, estará mucho mejor si encuentra todo el tiempo posible para escribir código, así que ¿por qué no empezar ahora?

**Preguntas**

1. Escriba un programa que llame a `fork()`. Antes de llamar a `fork()`, haga que el proceso principal acceda a una variable (por ejemplo, `x`) y establezca su valor en algo (por ejemplo, 100). ¿Qué valor tiene la variable en el proceso hijo?  
¿Qué sucede con la variable cuando tanto el hijo como el padre cambian el valor de `x`?
2. Escriba un programa que abra un archivo (con la llamada al sistema `open()`) y luego llame a `fork()` para crear un nuevo proceso. ¿Pueden tanto el niño como el padre acceder al descriptor de archivo devuelto por `open()`? ¿Qué sucede cuando escriben en el archivo al mismo tiempo, es decir, al mismo tiempo?
3. Escriba otro programa usando `fork()`. El proceso hijo debería imprimir "hola"; el proceso padre debería imprimir "adiós". Debe intentar asegurarse de que el proceso hijo siempre se imprima primero; ¿Puedes hacer esto sin llamar a `wait()` en el padre?
4. Escriba un programa que llame a `fork()` y luego llame a alguna forma de `exec()` para ejecutar el programa `/bin/ls`. Vea si puede probar todas las variantes de `exec()`, incluidas (en Linux) `execl()`, `execl()`, `execlp()`, `execv()`, `execvp()` y `execvpe()`. ¿Por qué crees que existen tantas variantes de una misma llamada básica?
5. Ahora escriba un programa que use `wait()` para esperar a que finalice el proceso hijo en el padre. ¿Qué devuelve `esperar()`? ¿Qué sucede si usas `wait()` en el niño?

6. Escriba una ligera modificación del programa anterior, esta vez usando waitpid() en lugar de wait(). ¿Cuándo sería útil waitpid() ?
7. Escriba un programa que cree un proceso hijo y luego en el hijo cierre la salida estándar (STDOUT\_FILENO). ¿Qué sucede si el niño llama a printf() para imprimir algún resultado después de cerrar el descriptor?
8. Escriba un programa que cree dos hijos y conecte la salida estándar de uno a la entrada estándar del otro, usando la llamada al sistema pipe() .

## Mecanismo: Ejecución Directa Limitada

Para virtualizar la CPU, el sistema operativo necesita compartir de alguna manera la CPU física entre muchos trabajos que aparentemente se ejecutan al mismo tiempo. La idea básica es simple: ejecutar un proceso por un tiempo, luego ejecutar otro, y así sucesivamente. Al compartir el tiempo de la CPU de esta manera, se logra la virtualización.

Sin embargo, existen algunos desafíos a la hora de construir dicha maquinaria de virtualización. El primero es el rendimiento: ¿cómo podemos implementar la virtualización sin añadir una sobrecarga excesiva al sistema? El segundo es el control: ¿cómo podemos ejecutar procesos de manera eficiente manteniendo el control sobre la CPU? El control es particularmente importante para el sistema operativo, ya que está a cargo de los recursos; sin control, un proceso podría simplemente ejecutarse para siempre y apoderarse de la máquina, o acceder a información a la que no se le debería permitir acceder. Por lo tanto, obtener un alto rendimiento manteniendo el control es uno de los desafíos centrales en la construcción de un sistema operativo.

### EL CRUCERO:

#### CÓMO VIRTUALIZAR EFICIENTEMENTE LA CPU CON CONTROL

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

### 6.1 Técnica básica: ejecución directa limitada

Para hacer que un programa se ejecute tan rápido como cabría esperar, no es sorprendente que los desarrolladores de sistemas operativos idearan una técnica que llamamos ejecución directa limitada. La parte de la idea de "ejecución directa" es simple: simplemente ejecute el programa directamente en la CPU. Por lo tanto, cuando el sistema operativo desea iniciar la ejecución de un programa, crea una entrada de proceso para él en una lista de procesos, le asigna algo de memoria, carga el código del programa en la memoria (desde el disco), localiza su punto de entrada (es decir, la rutina main() o algo similar), salta

SO	Programa
Crear entrada para la lista de procesos	
Asignar memoria para el programa.	
Cargar programa en memoria	
Configurar la pila con argc/argv	
Borrar registros	Ejecutar principal()
Ejecutar llamada principal()	Ejecutar retorno desde principal
Memoria libre de proceso	
Eliminar de la lista de procesos	

Figura 6.1: Protocolo de ejecución directa (sin límites)

y comienza a ejecutar el código del usuario. La Figura 6.1 muestra este protocolo básico de ejecución directa (sin límites todavía), usando una llamada normal y regresar para saltar al main() del programa y luego regresar al kernel.

Suena simple, ¿no? Pero este enfoque genera algunos problemas en nuestra búsqueda de virtualizar la CPU. La primera es simple: si simplemente ejecutamos un programa, ¿cómo puede el sistema operativo asegurarse de que el programa no haga nada que no queramos y al mismo tiempo ejecutarlo de manera eficiente? La segunda: cuando estamos ejecutando un proceso, ¿cómo el sistema operativo detiene su ejecución y cambia a otro proceso, implementando así el tiempo compartido que requerimos para virtualizar la CPU?

Al responder estas preguntas a continuación, tendremos una idea mucho mejor 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 sistema operativo no tendría control de nada y, por lo tanto, sería "sólo una biblioteca", ¡una situación muy triste para un aspirante a sistema operativo!

## 6.2 Problema n.<sup>o</sup> 1: operaciones restringidas

La ejecución directa tiene la ventaja obvia de ser rápida; el programa se ejecuta de forma nativa en la CPU del hardware y, por lo tanto, se ejecuta tan rápido como cabría esperar. Pero ejecutarlo en la CPU presenta un problema: ¿qué pasa si el proceso desea realizar algún tipo de operación restringida, como emitir una solicitud de E/S a un disco u obtener acceso a más recursos del sistema, como la CPU o la memoria?

### EL CRUX : CÓMO REALIZAR OPERACIONES RESTRINGIDAS

Un proceso debe poder realizar E/S y algunas otras operaciones restringidas, pero sin darle al proceso un control completo sobre el sistema.

¿Cómo pueden el sistema operativo y el hardware trabajar juntos para lograrlo?

## APARTE: POR QUÉ LAS LLAMADAS AL SISTEMA PARECEN LLAMADAS A PROCEDIMIENTOS

Quizás se pregunte por qué una llamada a una llamada al sistema, como open() o read(), se ve exactamente como una llamada a un procedimiento típico en C; es decir, si parece una llamada a un procedimiento, ¿cómo sabe el sistema que es una llamada al sistema y hace todo lo correcto? La razón simple: es una llamada a procedimiento, pero oculta dentro de esa llamada a procedimiento está la famosa instrucción trampa. Más específicamente, cuando llamas a open() (por ejemplo), estás ejecutando una llamada a un procedimiento en la biblioteca C. Allí, ya sea para open() o cualquiera de las otras llamadas al sistema proporcionadas, la biblioteca utiliza una convención de llamada acordada con el kernel para colocar los argumentos de open() en ubicaciones bien conocidas (por ejemplo, en la pila, o en registros específicos), también coloca el número de llamada del sistema en una ubicación conocida (nuevamente, en la pila o en un registro) y luego ejecuta la instrucción trap antes mencionada. El código de la biblioteca después de la trampa descomprime los valores de retorno y devuelve el control al programa que emitió la llamada al sistema. Por lo tanto, las partes de la biblioteca C que realizan llamadas al sistema están codificadas manualmente en ensamblador, ya que necesitan seguir cuidadosamente las convenciones para procesar argumentos y devolver valores correctamente, así como ejecutar la instrucción trap específica del hardware. Y ahora sabe por qué usted personalmente no tiene que escribir código ensamblador para integrarlo en un sistema operativo; alguien ya ha escrito esa asamblea para usted.

Un enfoqueería simplemente dejar que cualquier proceso haga lo que quiera en términos de E/S y otras operaciones relacionadas. Sin embargo, hacerlo impediría 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 proceso de usuario emita E/S al disco; si lo hicieramos, un proceso podría simplemente leer o escribir todo el disco y así se perderían todas las protecciones.

Por tanto, el enfoque que adoptamos es introducir un nuevo modo de procesador, conocido como modo de usuario; El código que se ejecuta en modo de usuario está restringido en lo que puede hacer. Por ejemplo, cuando se ejecuta en modo de usuario, un proceso no puede emitir solicitudes de E/S; hacerlo daría como resultado que el procesador generara una excepción; Entonces el sistema operativo probablemente finalizaría el proceso.

A diferencia del modo usuario, está el modo kernel, en el que se ejecuta el sistema operativo (o kernel). En este modo, el código que se ejecuta puede hacer lo que quiera, incluidas operaciones privilegiadas como emitir solicitudes de E/S y ejecutar todo tipo de instrucciones restringidas.

Sin embargo, todavía nos queda un desafío: ¿qué debe hacer un proceso de usuario cuando desea realizar algún tipo de operación privilegiada, como leer desde un disco? Para permitir esto, prácticamente todo el hardware moderno brinda a los programas de usuario la capacidad de realizar una llamada al sistema.

Iniciadas en máquinas antiguas como Atlas [K+61,L78], las llamadas al sistema permiten al kernel exponer cuidadosamente ciertas piezas clave de funcionalidad a los programas del usuario, como acceder al sistema de archivos, crear y destruir procesos, comunicarse con otros. procesos y asignar más

**CONSEJO: UTILICE LA TRANSFERENCIA DE CONTROL**

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

memoria. La mayoría de los sistemas operativos proporcionan unos cientos de llamadas (consulte el estándar POSIX para obtener más detalles [P10]); Los primeros sistemas Unix expusieron un subconjunto más conciso de alrededor de veinte llamadas.

Para ejecutar una llamada al sistema, un programa debe ejecutar una instrucción de captura especial. Esta instrucción salta simultáneamente al kernel y eleva el nivel de privilegio al modo kernel; Una vez en el kernel, el sistema ahora puede realizar cualquier operación privilegiada que sea necesaria (si está permitida) y así realizar el trabajo requerido para el proceso de llamada. Cuando termina, el sistema operativo llama a una instrucción especial de retorno desde trampa que, como es de esperar, regresa al programa de usuario que realiza la llamada y al mismo tiempo reduce el nivel de privilegios al modo de usuario.

El hardware debe tener un poco de cuidado al ejecutar una trampa, ya que debe asegurarse de guardar suficientes registros de la persona que llama para poder regresar correctamente cuando el sistema operativo emita la instrucción de retorno de la trampa. En x86, por ejemplo, el procesador enviará el contador del programa, las banderas y algunos otros registros a una pila del kernel por proceso; el return-from-trap sacará estos valores de la pila y reanudará la ejecución del programa en modo de usuario (consulte los manuales de sistemas Intel [I11] para obtener 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 trampa qué código ejecutar dentro del sistema operativo? Claramente, el proceso de 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 núcleo, lo que claramente es una muy mala idea<sup>1</sup>. Por tanto, el núcleo debe controlar cuidadosamente qué código se ejecuta en una trampa.

El kernel lo hace configurando una tabla de trampas en el momento del arranque. Cuando la máquina arranca, lo hace en modo privilegiado (kernel) y, por lo tanto, puede configurar el hardware de la máquina según sea necesario. Una de las primeras cosas que hace el sistema operativo es decirle al hardware qué código ejecutar cuando ocurren ciertos eventos excepcionales. Por ejemplo, ¿qué código debería 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 del

---

<sup>1</sup>Imagínese saltar al código para acceder a un archivo, pero justo después de una verificación de permiso; de hecho, es probable que dicha capacidad permita a un programador astuto conseguir que el núcleo ejecute secuencias de código arbitrarias [S07]. En general, trate de evitar ideas muy malas como ésta.

## MECANISMO: EJECUCIÓN DIRECTA LIMITADA

5

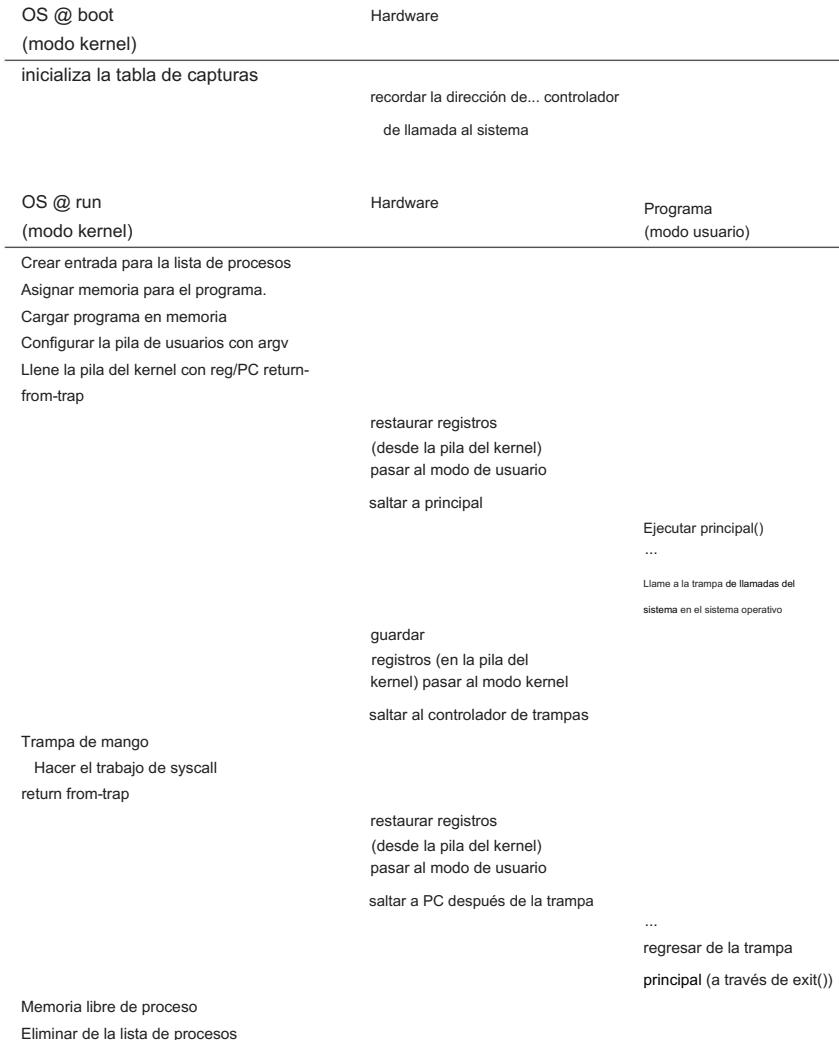


Figura 6.2: Protocolo de ejecución directa limitada

ubicaciones de estos manipuladores de trampas, generalmente con algún tipo de instrucción especial. Una vez que se informa al hardware, recuerda la ubicación de estos controladores hasta que se reinicia la máquina y, por lo tanto, el hardware sabe qué hacer (es decir, a qué código saltar) cuando se producen llamadas al sistema y otros eventos excepcionales.

## CONSEJO: TENGA CUIDADO CON LAS ENTRADAS DEL USUARIO EN

SISTEMAS SEGUROS Aunque nos hemos esforzado mucho en proteger el sistema operativo durante las llamadas al sistema (agregando un mecanismo de captura de hardware y asegurándonos de que todas las llamadas al sistema operativo se enruten a través de él), todavía hay muchas otras 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 sistema operativo debe verificar lo que ingresa el usuario y asegurarse de que los argumentos estén especificados correctamente o, de lo contrario, 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 (ya sea accidental o maliciosamente) pasa una dirección “mala” (por ejemplo, una dentro de la porción del espacio de direcciones del núcleo), el sistema operativo debe detectarlo y rechazar la llamada.

De lo contrario, un usuario podría leer toda la memoria del kernel; Dado que la memoria kernel (virtual) también suele incluir toda la memoria física del sistema, este pequeño error permitiría a un programa leer la memoria de cualquier otro proceso en el sistema.

En general, un sistema seguro debe tratar las entradas de los usuarios con gran sospecha. No hacerlo conducirá sin duda a un software fácilmente pirateado, a una sensación desesperada de que el mundo es un lugar inseguro y aterrador, y a la pérdida de seguridad laboral para el desarrollador de sistemas operativos que confía demasiado.

Para especificar la llamada al sistema exacta, normalmente se asigna un número de llamada al sistema a cada llamada al sistema. Por tanto, el código de usuario es responsable de colocar el número de llamada del sistema deseado en un registro o en una ubicación específica de la pila; El sistema operativo, cuando maneja la llamada al sistema dentro del controlador de trampas, examina este número, se asegura de que sea válido y, si lo es, ejecuta el código correspondiente. Este nivel de direccionamiento indirecto sirve como 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.

Un último comentario: poder ejecutar la instrucción para indicarle al hardware dónde están las tablas de captura es una capacidad muy poderosa. Por tanto, como habrás adivinado, también es una operación privilegiada . Si intenta ejecutar esta instrucción en modo usuario, el hardware no le permitirá y probablemente pueda adivinar lo que sucederá (pista: adiós, programa ofensivo).

Punto para reflexionar: ¿qué cosas horribles podrías hacerle a un sistema si pudieras instalar tu propia mesa trampa? ¿Podrías hacerte cargo 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 una pila del kernel donde los registros (incluidos los registros de propósito general y el contador del programa) se guardan y restauran (mediante el hardware) durante la transición dentro y fuera del kernel.

Hay dos fases en el protocolo de ejecución directa limitada (LDE) .

En el primero (en el momento del arranque), el kernel inicializa la tabla de trampas y la CPU recuerda su ubicación para su uso posterior. El núcleo lo hace mediante una instrucción privilegiada (todas las instrucciones privilegiadas están resaltadas en negrita).

En el segundo (cuando se ejecuta un proceso), el núcleo configura algunas cosas (por ejemplo, asignar un nodo en la lista de procesos, asignar memoria) antes de usar una instrucción de retorno desde trampa para iniciar la ejecución del proceso; esto cambia la CPU al modo de usuario y comienza a ejecutar el proceso.

Cuando el proceso desea emitir una llamada al sistema, regresa al sistema operativo, que lo maneja y una vez más devuelve el control al proceso mediante un retorno desde la trampa. Luego, el proceso completa su trabajo y regresa de main(); esto generalmente regresará a algún código auxiliar que saldrá correctamente del programa (por ejemplo, llamando a la llamada al sistema exit() , que queda atrapada en el sistema operativo). En este punto, el sistema operativo se limpia y listo.

## 6.3 Problema nº 2: Cambio entre procesos

El siguiente problema con la ejecución directa es lograr un cambio entre procesos. Cambiar entre procesos debería ser sencillo, ¿verdad? El sistema operativo debería simplemente decidir detener un proceso e iniciar otro. ¿Cuál es el problema? Pero en realidad es un poco complicado: específicamente, si un proceso se está ejecutando en la CPU, esto por definición significa que el sistema operativo no se está ejecutando. Si el sistema operativo 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 sistema operativo realice una acción si no se está ejecutando en la CPU. Llegamos así al meollo del problema.

### EL CRUX: CÓMO RECUPERAR EL CONTROL DE LA CPU

¿Cómo puede el sistema operativo recuperar el control de la CPU para poder cambiar entre procesos?

Un enfoque cooperativo: esperar llamadas al sistema Un enfoque que

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

Por lo tanto, podría preguntarse, ¿cómo es posible que un proceso amigable renuncie a la CPU en este mundo utópico? Resulta que la mayoría de los procesos transfieren el control de la CPU al sistema operativo con bastante frecuencia realizando llamadas al sistema, por ejemplo, para abrir un archivo y luego leerlo, o enviar un mensaje a otra máquina, o crear un nuevo proceso. . Sistemas como este a menudo incluyen una llamada explícita al sistema de rendimiento , que no hace más que transferir el control al sistema operativo para que pueda ejecutar otros procesos.

Las aplicaciones también transfieren el control al sistema operativo cuando hacen algo ilegal. Por ejemplo, si una aplicación se divide por cero o intenta acceder a una memoria a la que no debería poder acceder, generará una trampa para el

SO. El sistema operativo volverá a tener control de la CPU (y probablemente finalizará el proceso infractor).

Por lo tanto, en un sistema de programación cooperativo, el sistema operativo recupera el control de la CPU esperando que se realice una llamada al sistema o una operación ilegal de algún tipo. Quizás también estés 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 entonces el sistema operativo?

Un enfoque no cooperativo: el sistema operativo toma el control Sin alguna ayuda

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

#### EL CRUZ: CÓMO OBTENER EL CONTROL SIN COOPERACIÓN ¿Cómo puede el sistema

operativo obtener el control de la CPU incluso si los procesos no cooperan? ¿Qué puede hacer el sistema operativo para garantizar que un proceso no autorizado no se apodere de la máquina?

La respuesta resulta sencilla y fue descubierta hace muchos años por varias personas que fabricaban sistemas informáticos: una interrupción del temporizador [M+63]. Se puede programar un dispositivo temporizador para generar una interrupción cada cierto número de milisegundos; cuando se genera la interrupción, el proceso que se está ejecutando actualmente se detiene y se ejecuta un controlador de interrupciones preconfigurado en el sistema operativo.

En este punto, el sistema operativo ha recuperado el control de la CPU y, por lo tanto, puede hacer lo que quiera: detener el proceso actual e iniciar uno diferente.

Como comentamos antes con las llamadas al sistema, el sistema operativo debe informar al hardware qué código ejecutar cuando se produce la interrupción del temporizador; por lo tanto, en el momento del arranque, el sistema operativo hace exactamente eso. En segundo lugar, también durante la secuencia de inicio, el sistema operativo debe iniciar el temporizador, que por supuesto es un privilegio.

#### CONSEJO: TRATAR EL MAL COMPORTAMIENTO EN LA APLICACIÓN

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 sistema operativo intenta manejar este tipo de conducta ilícita es simplemente eliminar al infractor. ¡Un strike y estás fuera! Quizás brutal, pero ¿qué más debería hacer el sistema operativo cuando intentas acceder a la memoria ilegalmente o ejecutar una instrucción ilegal?

operación. Una vez que el cronómetro ha comenzado, el sistema operativo puede sentirse seguro de que eventualmente se le devolverá el control y, por lo tanto, el sistema operativo puede ejecutar programas de usuario. El temporizador también se puede apagar (también es una operación privilegiada), algo que discutiremos más adelante cuando comprendamos la concurrencia con más detalle.

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

#### Guardar y restaurar el contexto Ahora que el

sistema operativo ha recuperado el control, ya sea de forma cooperativa a través de una llamada al sistema o de manera más contundente a través de una interrupción del temporizador, se debe tomar una decisión: si continuar ejecutando el proceso que se está ejecutando actualmente o cambiar a un uno diferente. Esta decisión la toma una parte del sistema operativo conocida como programador; Analizaremos las políticas de programación con gran detalle en los próximos capítulos.

Si se toma la decisión de cambiar, el sistema operativo ejecuta un fragmento de código de bajo nivel al que nos referimos como cambio de contexto. Un cambio de contexto es conceptualmente simple: todo lo que el sistema operativo tiene que hacer es guardar algunos valores de registro para el proceso que se está ejecutando actualmente (en su pila del kernel, por ejemplo) y restaurar algunos para el proceso que pronto se ejecutará (de su pila de kernel). Al hacerlo, el sistema operativo garantiza que cuando finalmente se ejecute la instrucción de retorno desde trampa, 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 sistema operativo ejecutará algún código ensamblador de bajo nivel para guardar los registros de propósito general, la PC y el puntero de la pila del kernel del proceso que se está ejecutando actualmente, y luego restaurará dicho proceso. registros, PC y cambie a la pila del kernel para el proceso que pronto se ejecutará. Al cambiar de pila, el kernel ingresa 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 sistema operativo finalmente ejecuta una instrucción de retorno desde trampa,

#### CONSEJO: UTILICE LA INTERRUPCIÓN DEL TEMPORIZADOR PARA RECUPERAR EL CONTROL

La adición de una interrupción del temporizador le da al sistema operativo 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 sistema operativo a mantener el control de la máquina.

**CONSEJO: REINICIAR ES ÚTIL**

Anteriormente, señalamos que la única solución para los bucles infinitos (y comportamientos similares) bajo la preferencia cooperativa es reiniciar la máquina. Si bien es posible que se burle de este truco, los investigadores han demostrado que reiniciar (o, en general, reiniciar algún software) puede ser una herramienta enormemente útil para construir sistemas robustos [C+04].

Especificamente, reiniciar es útil porque devuelve el software a un estado conocido y probablemente más probado. Los reinicios también recuperan recursos obsoletos o filtrados (por ejemplo, memoria) que de otro modo podrían ser difíciles de manejar. Finalmente, los reinicios son fáciles de automatizar. Por todas estas razones, no es raro que en los servicios de Internet de clústeres a gran escala el software de administración de sistemas reinicie periódicamente conjuntos de máquinas para restablecerlas y así obtener las ventajas enumeradas anteriormente.

Por lo tanto, la próxima vez que reinicies, no estarás simplemente implementando algún truco feo. Más bien, está utilizando un enfoque probado para mejorar el comportamiento de un sistema informático. ¡Bien hecho!

el proceso que pronto se ejecutará se convierte en el proceso que se está ejecutando actualmente. Y así se completa el cambio de contexto.

En la Figura 6.3 se muestra un cronograma de todo el proceso. En este ejemplo, el proceso A se está ejecutando y luego es interrumpido por la interrupción del temporizador. El hardware guarda sus registros (en su pila de kernel) e ingresa al kernel (cambiando al modo kernel). En el controlador de interrupciones del temporizador, el sistema operativo 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 Proceso B (desde su entrada de estructura de proceso) y luego cambia de contexto, específicamente cambiando el puntero de la pila para usar la pila del núcleo de B (y no la de A). Finalmente, el sistema operativo regresa de la trampa, lo que restaura los registros de B y comienza a ejecutarlo.

Tenga en cuenta que hay dos tipos de guardados/restauraciones de registros que ocurren durante este protocolo. La primera es cuando se produce la interrupción del temporizador; en este caso, los registros de usuario del proceso en ejecución son guardados implícitamente por el hardware, utilizando la pila del kernel de ese proceso. La segunda es cuando el sistema operativo decide cambiar de A a B; en este caso, los registros del kernel son guardados explícitamente por el software (es decir, el sistema operativo), pero esta vez en la memoria en la estructura del proceso. La última acción hace que el sistema pase de ejecutarse como si acabara de quedar atrapado en el núcleo desde A a como si acabara de quedar atrapado en el núcleo desde B.

Para darle una mejor idea de cómo se implementa dicho cambio, la Figura 6.4 muestra el código de cambio de contexto para xv6. Vea si puede entenderlo (tendrá que saber un poco de x86, así como algo de xv6, para hacerlo). Las estructuras de contexto antiguas y nuevas se encuentran en las estructuras de proceso del proceso antiguo y nuevo, respectivamente.

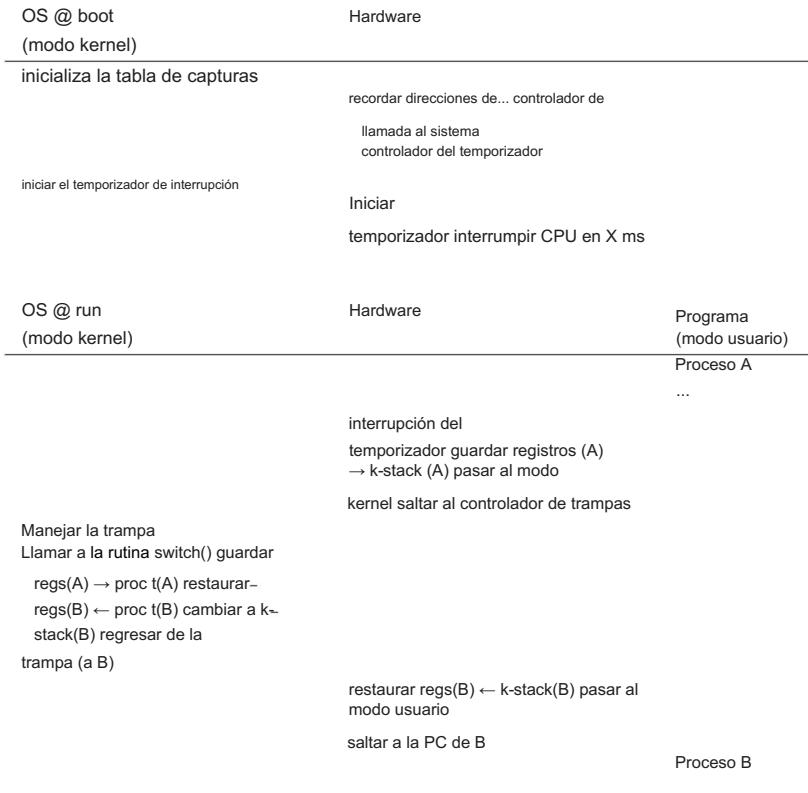


Figura 6.3: Protocolo de ejecución directa limitada (interrupción del temporizador)

## 6.4 ¿Le preocupa la simultaneidad?

Algunos de ustedes, como lectores atentos y reflexivos, pueden estar pensando ahora: "Hmm... ¿qué sucede cuando, durante una llamada al sistema, se produce una interrupción del temporizador?" o "¿Qué sucede cuando estás manejando una interrupción y ocurre otra? ¿No resulta difícil de manejar en el núcleo? Buenas preguntas: ¡todavía tenemos algunas esperanzas para ti!

La respuesta es sí, el sistema operativo debe preocuparse por lo que sucede si, durante el manejo de interrupciones o trampas, ocurre otra interrupción.

Este, de hecho, es el tema exacto de toda la segunda parte de este libro, sobre la concurrencia; Aplazaremos una discusión detallada hasta entonces.

Para abrirle el apetito, simplemente esbozaremos algunos conceptos básicos de cómo el sistema operativo maneja estas situaciones difíciles. Una cosa sencilla que podría hacer un sistema operativo es desactivar las interrupciones durante el procesamiento de interrupciones; hacerlo garantiza que cuando

```

1 # interruptor vacío (contexto de estructura **antiguo, contexto de estructura *nuevo);
2 #
3 # Guardar el contexto del registro actual en el antiguo
4 # y luego cargar el contexto del registro desde nuevo.
5 .cambio global
6 interruptor:
7     # Guardar registros antiguos
8     movl 4(%esp), %eax # poner ptr antiguo en eax
9     popl 0(%eax) movl           # guardar la IP antigua
10    %esp, 4(%eax) # y pila
11    movl %ebx, 8(%eax) # y otros registros
12    movimiento %ecx, 12(%eax)
13    movimiento %edx, 16(%eax)
14    movimiento %esi, 20(%eax)
15    movimiento %edi, 24(%eax)
16    movimiento %ebp, 28(%eax)
17
18     # Cargar nuevos registros
19     movl 4(%esp), %eax # poner nuevo ptr en eax
20     movl 28(%eax), %ebp # restaurar otros registros
21     movimiento 24(%eax), %edi
22     movimiento 20(%eax), %esi
23     movimiento 16(%eax), %edx
24     movimiento 12(%eax), %ecx
25     movimiento 8(%eax), %ebx
26     movl 4(%eax), %esp # la pila se cambia aquí
27     pushl 0(%eax) # dirección de retorno implementada
28     ret # finalmente regresa al nuevo ctx

```

Figura 6.4: El código de cambio de contexto xv6

Se está manejando una interrupción, no se entregará ninguna otra a la CPU.

Por supuesto, el sistema operativo debe tener cuidado al hacerlo; deshabilitar interrupciones para demasiado tiempo podría provocar la pérdida de interrupciones, lo cual es (en términos técnicos) malo.

Los sistemas operativos también han desarrollado una serie de sofisticados esquemas de bloqueo para proteger el acceso simultáneo a estructuras de datos internas. Esto permite que múltiples actividades estén en curso dentro del kernel al mismo tiempo. Al mismo tiempo, particularmente útil en multiprocesadores. Como veremos en el Sin embargo, en la siguiente 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 descrito algunos mecanismos clave de bajo nivel para implementar CPU virtualización, un conjunto de técnicas a las que colectivamente nos referimos como limitadas. ejecución directa. La idea básica es sencilla: simplemente ejecute el programa. desea ejecutar en la CPU, pero primero asegúrese de configurar el hardware para limitar lo que el proceso puede hacer sin la ayuda del sistema operativo.

**APARTE: ¿ CUÁNTO TIEMPO TARDAN LOS CAMBIOS DE CONTEXTO ? Una**

pregunta natural que podrías tener es: ¿cuánto tiempo lleva algo así como un cambio de contexto? ¿O incluso una llamada al sistema? Para aquellos que tengan curiosidad, existe una herramienta llamada Imbech [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 en un orden de magnitud mejor, con resultados inferiores a microsegundos en sistemas con procesadores de 2 o 3 GHz.

Cabe señalar que no todas las acciones del sistema operativo realizan un seguimiento del rendimiento de la CPU. Como observó Ousterhout, muchas operaciones del sistema operativo requieren mucha memoria y el ancho de banda de la memoria no ha mejorado tan dramáticamente como la velocidad del procesador con el tiempo [O90]. Por lo tanto, dependiendo de su carga de trabajo, es posible que comprar el último y mejor procesador no acelere su sistema operativo tanto como podría esperar.

Este enfoque general también se adopta en la vida real. Por ejemplo, aquellos de ustedes que tienen hijos, o, al menos, han oído hablar de niños, pueden estar familiarizados con el concepto de hacer una habitación a prueba de bebés : cerrar con llave los gabinetes que contienen cosas peligrosas y cubrir los enchufes eléctricos. Cuando la habitación esté preparada, podrá dejar que su bebé deambule libremente, sabiendo que los aspectos más peligrosos de la habitación han sido restringidos.

De manera análoga, el sistema operativo pone a prueba la CPU, configurando primero (durante el arranque) los controladores de trampas e iniciando un temporizador de interrupción, y luego ejecutando procesos únicamente en un modo restringido. Al hacerlo, el sistema operativo puede sentirse bastante seguro de que los procesos pueden ejecutarse de manera eficiente y solo requieren la intervención del sistema operativo para realizar operaciones privilegiadas o cuando han monopolizado la CPU durante demasiado tiempo y, por lo tanto, es necesario desconectarlos.

Disponemos así de los mecanismos básicos para virtualizar la CPU. Pero queda sin respuesta una pregunta importante: ¿qué proceso deberíamos ejecutar en un momento dado? Es esta pregunta la que debe responder el planificador y, por tanto, el siguiente tema de nuestro estudio.

**ADEMÁS: TÉRMINOS CLAVE DE VIRTUALIZACIÓN DE LA CPU (MECANISMOS)**

- La CPU debe admitir al menos dos modos de ejecución: un modo de usuario restringido y un modo de kernel privilegiado (no restringido). • Las aplicaciones de usuario típicas se ejecutan en modo de usuario y utilizan una llamada al sistema para ingresar 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 preespecificado: la tabla trap.
- Cuando el sistema operativo termina de atender una llamada al sistema, regresa al programa de usuario a través de otra instrucción especial de retorno desde trampa , que reduce el privilegio y devuelve el control a la instrucción después de la trampa que saltó al sistema operativo.
- Las tablas de captura deben ser configuradas por el sistema operativo en el momento del arranque 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 pérdida de control del sistema operativo.
- Una vez que un programa se está ejecutando, el sistema operativo debe utilizar mecanismos de hardware para garantizar que el programa del usuario no se ejecute eternamente, es decir, la interrupción del temporizador. Este enfoque es un enfoque no cooperativo para la programación de la CPU.
- A veces, el sistema operativo, durante una interrupción del temporizador o una llamada al sistema, puede desear pasar del proceso actual a otro diferente, una técnica de bajo nivel conocida como cambio de contexto.

## Referencias

- [A79] "Manual del usuario de Alto" de Xerox. Centro de Investigación Xerox Palo Alto, septiembre de 1979. Disponible: <http://history-computer.com/Library/AltoUsersHandbook.pdf>. Un sistema asombroso, muy adelantado a su tiempo. Se hizo famoso porque Steve Jobs la visitó, tomó notas y construyó Lisa y, finalmente, Mac.
- [C+04] "Microreboot: una técnica para una recuperación económica" por G. Candeia, S. Kawamoto, Y. Fujiki, G. Friedman, A. Fox. OSDI '04, San Francisco, CA, diciembre de 2004. Un excelente artículo que señala hasta dónde se puede llegar con el reinicio en la construcción de sistemas más robustos.
- [I11] "Manual del desarrollador de software de arquitecturas Intel 64 e IA-32" por Volumen 3A y 3B: Guía de programación del sistema. Intel Corporation, enero de 2011. Este es simplemente un manual aburrido, pero a veces resulta útil.
- [K+61] "Sistema de almacenamiento de un nivel" por 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. Sin embargo, este artículo no es el mejor para leer. Si solo leyeras uno, podrías probar la perspectiva histórica a continuación [L78].
- [L78] "El Manchester Mark I y el Atlas: una perspectiva histórica" por SH Lavington. Communications of the ACM, 21:1, enero de 1978. Una historia del desarrollo inicial de las computadoras y los esfuerzos pioneros de Atlas.
- [M+63] "Un sistema de depuración de tiempo compartido para una computadora pequeña" por J. McCarthy, S. Boilen, E. Fredkin, JCR Licklider. AFIPS '63 (primavera), mayo de 1963, Nueva York, Estados Unidos. Uno de los primeros artículos sobre el tiempo compartido que se refiere al uso de una interrupción del temporizador; la cita que lo analiza: "La tarea básica de la rutina de reloj del canal 17 es decidir si se elimina al usuario actual del núcleo y, de ser así, decidir qué programa de usuario intercambiar cuando salga".
- [MS96] "lmbench: herramientas portátiles para análisis de rendimiento" por Larry McVoy y Carl Staelin. Conferencia técnica anual de USENIX, enero de 1996. Un documento divertido sobre cómo medir diferentes aspectos de su sistema operativo y su rendimiento. Descarga lmbench y pruébalo.
- [M11] "Mac OS 9" de Apple Computer, Inc. Enero de 2011. [http://en.wikipedia.org/wiki/Mac\\_OS\\_9](http://en.wikipedia.org/wiki/Mac_OS_9) . Probablemente incluso puedes encontrar un emulador de OS 9 si lo deseas; ¡Compruébalo, es un pequeño Mac divertido!
- [O90] "¿Por qué los sistemas operativos no son cada vez más rápidos que el hardware?" por J. Ouster-hout. Conferencia de verano de USENIX, junio de 1990. Un artículo clásico sobre la naturaleza del rendimiento del sistema operativo.
- [P10] "La especificación única de UNIX, versión 3" de The Open Group, mayo de 2010. Disponible: <http://www.unix.org/version3/>. Esto es difícil y doloroso de leer, así que probablemente evítelo si puedes. A menos que alguien te pague por leerlo. ¡O simplemente tienes tanta curiosidad que no puedes evitarlo!
- [S07] "La geometría de la carne inocente en el hueso: regreso a libc sin llamadas a funciones (en el x86)" por Hovav Shacham. CCS '07, octubre de 2007. Una de esas ideas asombrosas y alucinantes que verás en las investigaciones de vez en cuando. El autor muestra que si puedes saltar al código de forma arbitraria, básicamente puedes unir cualquier secuencia de código que desees (dada una base de código grande); Lea el documento para conocer los detalles. Lamentablemente, esta técnica hace que sea aún más difícil defenderte contra ataques maliciosos.

## Tarea (medición)

### APARTE: TAREAS DE MEDICIÓN Las tareas de medición

son pequeños ejercicios en los que se escribe código para ejecutarlo 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 brindarle un poco de experiencia práctica con un sistema operativo real.

En esta tarea, medirá 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, podría realizar repetidamente una llamada simple al sistema (p. ej., realizar una lectura de 0 bytes) y cronometrar el tiempo que lleva; dividir el tiempo por el número de iteraciones le da una estimación del costo de una llamada al sistema.

Una cosa que tendrás que tener en cuenta es la precisión y exactitud de tu cronómetro. Un temporizador típico que puedes utilizar es `gettimeofday()`; lea la página de manual para obtener más detalles. Lo que verás allí es que `gettimeofday()` devuelve la hora en microsegundos desde 1970; sin embargo, esto no significa que el cronómetro tenga una precisión de microsegundos. Mida las llamadas consecutivas a `gettimeofday()` para aprender algo sobre qué tan preciso es realmente el temporizador; esto le indicará cuántas iteraciones de su prueba de llamada nula al sistema tendrá que ejecutar para obtener un buen resultado de medición. Si `gettimeofday()` no es lo suficientemente preciso para usted, puede considerar utilizar la instrucción `rdtsc` disponible en máquinas x86.

Medir el costo de un cambio de contexto es un poco más complicado. El benchmark `lmbench` lo hace ejecutando dos procesos en una sola CPU y configurando dos canalizaciones UNIX entre ellos; una tubería es sólo una de las muchas formas en que los procesos en un sistema UNIX pueden comunicarse entre sí. Luego, el primer proceso emite una escritura en el primer canal y espera una lectura en el segundo; al ver el primer proceso esperando a que se lea algo del segundo canal, el sistema operativo coloca el primer proceso en estado bloqueado y cambia al otro proceso, que lee del primer canal y luego escribe en el segundo. Cuando el segundo proceso intenta leer nuevamente desde la primera tubería, se bloquea y, por lo tanto, continúa el ciclo de comunicación de ida y vuelta. Al medir repetidamente el costo de comunicarse de esta manera, `lmbench` puede hacer una buena estimación del costo de un cambio de contexto. Puede intentar recrear algo similar aquí, utilizando tuberías 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 asegurarse de que ambos procesos estén en el mismo procesador, se asegura de medir el costo de que el sistema operativo detenga un proceso y restaure otro en la misma CPU.

## Programación: Introducción

A estas alturas los mecanismos de bajo nivel de ejecución de procesos (por ejemplo, cambio de contexto) deberían estar claros; si no es así, retroceda uno o dos capítulos y lea la descripción de cómo funcionan esas cosas nuevamente. Sin embargo, todavía tenemos que comprender las políticas de alto nivel que emplea un programador de sistema operativo. Ahora haremos precisamente eso, presentando una serie de políticas de programación (a veces llamadas disciplinas) que varias personas inteligentes y trabajadoras han desarrollado a lo largo de los años.

De hecho, los orígenes de la programación son anteriores a los sistemas informáticos; Los primeros enfoques se tomaron del campo de la gestión de operaciones y se aplicaron a las computadoras. Esta realidad no debería sorprender: las líneas de montaje y muchas otras actividades humanas también requieren programación, y en ellas existen muchas de las mismas preocupaciones, incluido un deseo de eficiencia.

Y así, nuestro problema:

### EL CRUX: CÓMO DESARROLLAR UNA POLÍTICA DE PROGRAMACIÓN

¿Cómo deberíamos desarrollar un marco básico para pensar en las políticas de programación? ¿Cuáles son los supuestos clave? ¿Qué métricas son importantes? ¿Qué enfoques básicos se han utilizado en los primeros sistemas informáticos?

### 7.1 Supuestos de carga de trabajo

Antes de entrar en la gama de políticas posibles, primero hagamos una serie de suposiciones simplificadoras sobre los procesos que se ejecutan en el sistema, a veces llamados colectivamente carga de trabajo. Determinar la carga de trabajo es una parte fundamental de la creación de políticas y cuanto más sepa sobre la carga de trabajo, más afinada podrá ser su política.

Las suposiciones sobre la carga de trabajo que hacemos aquí son en su mayoría poco realistas, pero está bien (por ahora), porque las relajaremos a medida que avancemos y eventualmente desarrollaremos lo que llamaremos... (pausa dramática)...

una disciplina de programación totalmente operativa<sup>1</sup>.

Haremos las siguientes suposiciones sobre los procesos, a veces llamados trabajos, que se ejecutan en el sistema:

1. Cada trabajo se ejecuta durante la misma cantidad de tiempo.
2. Todos los trabajos llegan al mismo tiempo.
3. Una vez iniciado, cada trabajo se ejecuta hasta su finalización.
4. Todos los trabajos sólo utilizan la CPU (es decir, no realizan E/S)
5. Se conoce el tiempo de ejecución de cada trabajo.

Dijimos que muchas de estas suposiciones no eran realistas, pero así como algunos animales son más iguales que otros en Animal Farm [O45] de Orwell, algunas suposiciones son menos realistas que otras en este capítulo. En particular, podría molestarle que se conozca el tiempo de ejecución de cada trabajo: esto haría que el programador fuera omnisciente, lo cual, aunque sería genial (probablemente), no es probable que suceda pronto.

## 7.2 Métricas de programación

Más allá de hacer suposiciones sobre la carga de trabajo, también necesitamos una cosa más que nos permita comparar diferentes políticas de programación: una métrica de programación. Una métrica es simplemente algo que usamos para medir algo, y hay varias métricas diferentes que tienen sentido en la programación.

Por ahora, sin embargo, simplifiquemos nuestra vida simplemente teniendo una única métrica: el tiempo de respuesta. El tiempo de respuesta de un trabajo se define como el tiempo en que se completa el trabajo menos el tiempo en que el trabajo llegó al sistema. Más formalmente, el tiempo de respuesta Turnaround es:

$$\text{Tturnaround} = \text{Tcompletado} - \text{Tarrival} \quad (7.1)$$

Porque hemos asumido que todos los trabajos llegan al mismo tiempo, por ahora Tarrival = 0 y por lo tanto Tturnaround = Tcompletion. Este hecho cambiará a medida que relajemos los supuestos antes mencionados.

Debe tener en cuenta que el tiempo de respuesta es una métrica de desempeño, que será nuestro enfoque principal en este capítulo. Otra métrica de interés es la equidad, medida (por ejemplo) por el Índice de Equidad de Jain [J91]. El desempeño y la equidad a menudo están reñidos en la programación; un programador, por ejemplo, puede optimizar el rendimiento, pero a costa de impedir que se ejecuten algunos trabajos, lo que reduce la equidad. Este enigma nos muestra que la vida no siempre es perfecta.

## 7.3 Primero en entrar, primero en salir (FIFO)

El algoritmo más básico que podemos implementar se conoce como programación primero en entrar, primero en salir (FIFO) o, a veces, por orden de llegada (FCFS).

---

<sup>1</sup>Dicho de la misma manera que dirías "Una Estrella de la Muerte en pleno funcionamiento".

FIFO tiene una serie de propiedades positivas: es claramente simple y por lo tanto fácil de implementar. Y, dadas nuestras suposiciones, funciona bastante bien.

Hagamos un ejemplo rápido juntos. Imaginemos que llegan tres trabajos al sistema, A, B y C, aproximadamente al mismo tiempo ( $T_{arrival} = 0$ ). Porque

FIFO tiene que poner algún trabajo primero, supongamos que si bien llegaron todos simultáneamente, A llegó apenas un pelo antes que B, que llegó apenas un pelo antes de C. Supongamos también que cada trabajo se ejecuta durante 10 segundos. ¿Qué será el ¿Cuál será el tiempo medio de respuesta para estos trabajos?

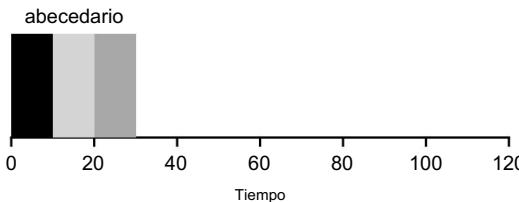


Figura 7.1: Ejemplo simple FIFO

En la Figura 7.1, puede ver que A terminó en 10, B en 20 y C en 30.

Por lo tanto, el tiempo de respuesta promedio para los tres trabajos es simplemente  $\frac{10+20+30}{3} = 20$ . Calcular el tiempo de respuesta es así de fácil.

Ahora relajemos una de nuestras suposiciones. En particular, relajemos el supuesto 1 y, por lo tanto, ya no supongamos que cada trabajo se ejecuta durante el mismo tiempo. cantidad de tiempo. ¿Cómo funciona FIFO ahora? ¿Qué tipo de carga de trabajo? ¿Podrías construir para que FIFO funcione mal?

(piensa en esto antes de seguir leyendo... sigue pensando... ¡¿entendido?!)

Probablemente ya lo hayas descubierto, pero por si acaso, hágámoslo. un ejemplo para mostrar cómo trabajos de diferente duración pueden generar problemas para Programación FIFO. En particular, supongamos nuevamente tres trabajos (A, B y C), pero esta vez A corre durante 100 segundos mientras que B y C corren durante 10 cada uno.

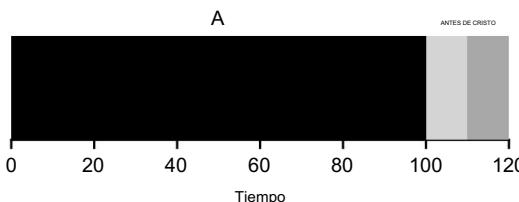


Figura 7.2: Por qué FIFO no es tan bueno

Como puede ver en la Figura 7.2, el trabajo A se ejecuta primero durante los 100 segundos completos. antes de que B o C tengan siquiera la oportunidad de correr. Por lo tanto, el retorno promedio

El tiempo para el sistema es alto: unos dolorosos 110 segundos (este  $\frac{100+110+120}{3} = 110$ ).

Este problema generalmente se conoce como efecto convoy [B+79], donde un número relativamente pequeño de consumidores potenciales de un recurso se ponen en cola

**CONSEJO: EL PRINCIPIO DE SJF**

El trabajo más corto primero representa un principio general de programación que puede aplicarse aplicado a cualquier sistema donde el tiempo de respuesta percibido por cliente (o, en nuestro caso, un trabajo) importa. Piensa en cualquier fila en la que hayas esperado: si el establecimiento en cuestión se preocupa por la satisfacción del cliente, es probable han tenido en cuenta a SJF. Por ejemplo, las tiendas de comestibles comúnmente tener una línea de "diez artículos o menos" para garantizar que los compradores con solo unos pocos cosas para comprar no se quede atrapado detrás de la familia preparándose para algo próximo invierno nuclear.

detrás de un gran consumidor de recursos. Este escenario de programación podría recordarte una sola fila en una tienda de comestibles y cómo te sientes cuando ves a la persona frente a ti con tres carros llenos de provisiones y una chequera afuera; Va a pasar un tiempo<sup>2</sup>.

Entonces, ¿qué debemos hacer? ¿Cómo podemos desarrollar un mejor algoritmo para Afrontar nuestra nueva realidad de trabajos que duran diferentes períodos de tiempo? Piénselo primero; entonces sigue leyendo.

#### 7.4 Primero el trabajo más corto (SJF)

Resulta que un enfoque muy sencillo resuelve este problema; En realidad es una idea robada de la investigación de operaciones [C54,PV56] y aplicada a programación de trabajos en sistemas informáticos. Esta nueva disciplina de programación se conoce como Trabajo más corto primero (SJF), y el nombre debería ser fácil de entender. recuerde porque describe la política de manera bastante completa: ejecuta el Primero el trabajo más corto, luego el siguiente más corto, y así sucesivamente.

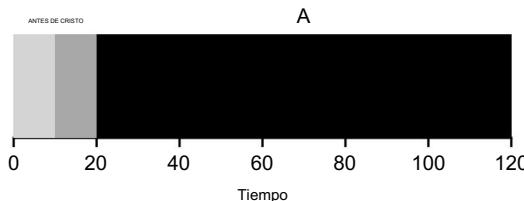


Figura 7.3: Ejemplo simple de SJF

Tomenos nuestro ejemplo anterior pero con SJF como nuestra política de programación. La figura 7.3 muestra los resultados de ejecutar A, B y C. Es de esperar que el diagrama aclare por qué SJF se desempeña mucho mejor con respecto al tiempo de respuesta promedio. Simplemente ejecutando B y C antes de A, SJF reduce respuesta promedio de 110 segundos a 50 ( $= 50$ ), más de  $\frac{10+20+120}{3}$  una mejora del factor dos.

<sup>2</sup>Acción recomendada en este caso: cambiar rápidamente a una línea diferente o tomar un tiempo largo y respiración profunda y relajante. Así es, inhale, exhale. Todo estará bien, no te preocupes.

## APARTE: PROGRAMADORES PREVENTARIOS

En los viejos tiempos de la computación por lotes, se desarrollaron varios programadores no preventivos ; tales sistemas ejecutarían cada trabajo hasta su finalización antes de considerar si ejecutar un nuevo trabajo. Prácticamente todos los programadores modernos son preventivos y están bastante dispuestos a detener la ejecución de un proceso para ejecutar otro. Esto implica que el planificador emplea el mecanismos que aprendimos anteriormente; en particular, el planificador puede realizar un cambio de contexto, deteniendo temporalmente un proceso en ejecución y reanudar (o iniciar) otro.

De hecho, dadas nuestras suposiciones acerca de que todos los empleos llegan al mismo tiempo, Podríamos demostrar que SJF es de hecho un algoritmo de programación óptimo . Sin embargo, estás en una clase de sistemas, no de teoría o investigación de operaciones; No Se permiten pruebas.

Así llegamos a un buen enfoque para programar con SJF, pero nuestro Las suposiciones siguen siendo bastante poco realistas. Relajemos otro. En particular, Podemos centrarnos en el supuesto 2 y ahora suponer que los empleos pueden llegar en cualquier momento. tiempo en lugar de todo a la vez. ¿A qué problemas conduce esto?

(Otra pausa para pensar... ¿estás pensando? Vamos, puedes hacerlo)

Aquí podemos ilustrar el problema nuevamente con un ejemplo. Esta vez, Supongamos que A llega a t = 0 y necesita correr durante 100 segundos, mientras que B y C llegan a t = 10 y cada uno necesita correr durante 10 segundos. con puro SJF, obtendríamos el cronograma que se ve en la Figura 7.4.

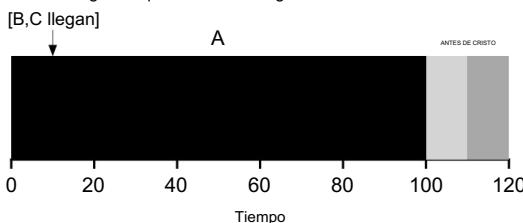


Figura 7.4: SJF con llegadas tardías desde B y C

Como puede ver en la figura, aunque B y C llegaron poco después después de A, todavía se ven obligados a esperar hasta que A haya completado y, por lo tanto, sufren el mismo problema de convoy. Tiempo promedio de respuesta para estos tres trabajos es  $\frac{100+(110-10)+(120-10)}{3} = 103,33$  segundos ( ). ¿Qué puede hacer un planificador?

## 7.5 Primero el tiempo de finalización más corto (STCF)

Para abordar esta preocupación, debemos relajar el supuesto 3 (que los empleos deben ejecutar hasta completar), así que hágámoslo. También necesitamos algo de maquinaria dentro el propio planificador. Como habrás adivinado, dada nuestra discusión anterior sobre interrupciones del temporizador y cambio de contexto, el programador puede

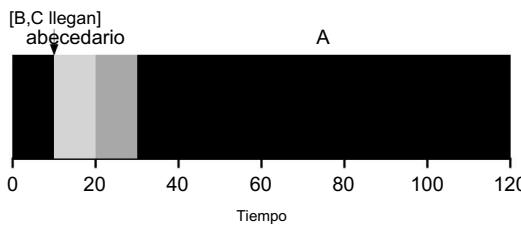


Figura 7.5: Ejemplo simple del STCF

ciertamente hacer algo más cuando lleguen B y C: puede adelantarse al trabajo A y decide ejecutar otro trabajo, tal vez continuar con A más adelante. SJF, según nuestra definición, es un programador no preventivo y, por lo tanto, sufre los problemas descrito anteriormente.

Afortunadamente, existe un programador que hace exactamente eso: agregar preferencia a SJF, conocido como el tiempo de finalización más corto primero (STCF) o Programador preventivo del trabajo más corto primero (PSJF) [CK68]. Cada vez que aparece un nuevo trabajo en el sistema, el programador STCF determina cuál de los trabajos restantes (incluido el nuevo trabajo) tiene menos tiempo restante y programa Aquél. Por lo tanto, en nuestro ejemplo, STCF se adelantaría a A y ejecutaría B y C, hasta su finalización; Sólo cuando hayan terminado, el tiempo restante de A será programado. La figura 7.5 muestra un ejemplo.

El resultado es un tiempo de respuesta medio muy mejorado: 50 segundos.  $(\frac{(120-0)+(20-10)+(30-10)}{3})$ . Y como antes, dadas nuestras nuevas suposiciones, STCF es demostrablemente óptimo; dado que SJF es óptimo si todos los trabajos llegan a Al mismo tiempo, probablemente deberías poder ver la intuición detrás la optimidad de STCF.

## 7.6 Una nueva métrica: tiempo de respuesta

Por lo tanto, si supiéramos la duración de los trabajos, y esos trabajos sólo utilizan la CPU, y nuestra La única métrica era el tiempo de respuesta, STCF sería una excelente política. De hecho, Para varios de los primeros sistemas informáticos por lotes, estos tipos de programación Los algoritmos tenían algún sentido. Sin embargo, la introducción del tiempo compartido Las máquinas cambiaron todo eso. Ahora los usuarios se sentarían frente a una terminal y exigirían también un rendimiento interactivo del sistema. Y así, una nueva Nació la métrica: el tiempo de respuesta.

Definimos el tiempo de respuesta como el tiempo desde que el trabajo llega a un sistema a la primera vez que se programa<sup>3</sup>. Más formalmente:

$$\text{Trespuesta} = \text{Tf} - \text{Tarrival} \quad (7.2)$$

<sup>3</sup> Algunos lo definen ligeramente diferente, por ejemplo, para incluir también el tiempo hasta que el trabajo produce algún tipo de "respuesta"; nuestra definición es la mejor versión de esto, asumiendo esencialmente que el trabajo produzca una respuesta instantánea.

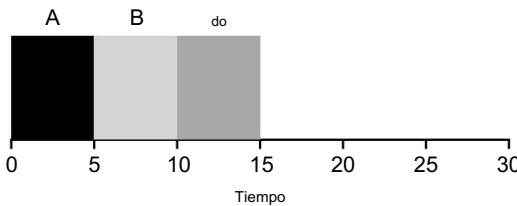


Figura 7.6: SJF nuevamente (malo para el tiempo de respuesta)

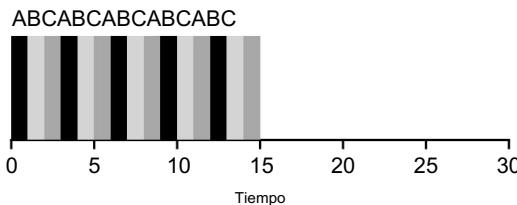


Figura 7.7: Round Robin (bueno para el tiempo de respuesta)

Por ejemplo, si tuviéramos el cronograma de la Figura 7.5 (con A llegando en el momento 0, y B y C en el momento 10), el tiempo de respuesta de cada trabajo es como sigue: 0 para el trabajo A, 0 para el B y 10 para el C (promedio: 3,33).

Como podría estar pensando, STCF y disciplinas relacionadas no son particularmente buenas para el tiempo de respuesta. Si llegan tres trabajos al mismo tiempo, por ejemplo, el tercer trabajo tiene que esperar a que se ejecuten los dos trabajos anteriores su totalidad antes de programarse solo una vez. Si bien es excelente para lograr cambios de tiempo, este enfoque es bastante malo para el tiempo de respuesta y la interactividad. De hecho, imagínese sentado frente a una terminal, escribiendo y teniendo que esperar 10 segundos. Ver una respuesta del sistema sólo porque se programó otro trabajo antes que el suyo: no es demasiado agradable.

Por tanto, nos queda otro problema: ¿cómo podemos construir un planificador? ¿Eso es sensible al tiempo de respuesta?

## 7.7 Ronda contra todos

Para resolver este problema, introduciremos un nuevo algoritmo de programación, clásicamente denominada programación Round-Robin (RR) [K64]. Lo básico La idea es simple: en lugar de ejecutar trabajos hasta su finalización, RR ejecuta un trabajo durante un segmento de tiempo (a veces llamado cuento de programación) y luego cambia al siguiente trabajo en la cola de ejecución. Lo hace repetidamente hasta que los trabajos están finalizado. Por esta razón, a RR a veces se le llama división de tiempo. Tenga en cuenta que la duración de un intervalo de tiempo debe ser un múltiplo del período de interrupción del temporizador; por lo tanto, si el temporizador se interrumpe cada 10 milisegundos, el intervalo de tiempo podría ser 10, 20 o cualquier otro múltiplo de 10 ms.

Para comprender RR con más detalle, veamos un ejemplo. Asumir tres trabajos A, B y C llegan al mismo tiempo al sistema, y que

**CONSEJO: LA AMORTIZACIÓN PUEDE REDUCIR COSTOS**

La técnica general de amortización se utiliza comúnmente en sistemas cuando existe un costo fijo para alguna operación. Al incurrir en ese costo menos A menudo (es decir, al realizar la operación menos veces), el costo total para el sistema se reduce. Por ejemplo, si el intervalo de tiempo se establece en 10 ms y el El costo del cambio de contexto es de 1 ms, aproximadamente el 10% del tiempo se dedica al cambio de contexto y, por lo tanto, se desperdicia. Si queremos amortizar este coste, podemos aumentar el intervalo de tiempo, por ejemplo, a 100 ms. En este caso, se dedica menos del 1% del tiempo. cambio de contexto y, por lo tanto, el costo de la división del tiempo se ha amortizado.

Cada uno de ellos desea correr durante 5 segundos. Un programador SJF ejecuta cada trabajo para finalización antes de ejecutar otro (Figura 7.6). En cambio, RR con un Un intervalo de tiempo de 1 segundo recorrería los trabajos rápidamente (Figura 7.7).

El tiempo medio de respuesta de RR es:  $0+1+2 = 1$ ; para SJF, promedio de re-  
el tiempo de respuesta es:  $0+\frac{5}{3}+10 = 5$ .

Como puede ver, la duración del intervalo de tiempo es fundamental para RR. el mas corto es decir, mejor será el rendimiento de RR según la métrica de tiempo de respuesta. Sin embargo, acortar demasiado el intervalo de tiempo es problemático: de repente, El costo del cambio de contexto dominará el rendimiento general. Por lo tanto, decidir sobre la duración del intervalo de tiempo presenta una compensación para el diseñador del sistema, ya que lo hace lo suficientemente largo como para amortizar el costo del cambio sin haciéndolo tan largo que el sistema ya no responde.

Tenga en cuenta que el costo del cambio de contexto no surge únicamente de la Acciones del sistema operativo para guardar y restaurar algunos registros. Cuando los programas se ejecutan, acumulan una gran cantidad de estado en cachés de CPU, TLB, predictores de rama, y otro hardware en chip. Cambiar a otro trabajo causa este estado para ser vaciado y nuevo estado relevante para el trabajo actualmente en ejecución incorporado, lo que puede suponer un coste de rendimiento notable [MB91].

Por lo tanto, RR, con un intervalo de tiempo razonable, es un excelente programador si el tiempo de respuesta es nuestra única métrica. Pero, ¿qué pasa con nuestro viejo amigo? ¿tiempo? Veamos nuevamente nuestro ejemplo anterior. A, B y C, cada uno con tiempos de ejecución de 5 segundos, llegan al mismo tiempo y RR es el planificador. con un intervalo de tiempo (largo) de 1 segundo. Podemos ver en la imagen de arriba que A termina en 13, B en 14 y C en 15, para un promedio de 14. ¡Bastante horrible!

No sorprende, entonces, que la GR sea de hecho una de las peores políticas si El tiempo de respuesta es nuestra métrica. Intuitivamente, esto debería tener sentido: ¿qué Lo que está haciendo RR es alargar cada trabajo tanto como pueda, ejecutando únicamente cada trabajo por un momento antes de pasar al siguiente. Porque el cambio al tiempo sólo le importa cuándo terminan los trabajos, el RR es casi pesimista, peor aún que el simple FIFO en muchos casos.

En términos más generales, cualquier política (como RR) que sea justa, es decir, que divida equitativamente la CPU entre procesos activos en una pequeña escala de tiempo, funcionará mal en métricas como el tiempo de respuesta. De hecho, esto es inherente compensación: si está dispuesto a ser injusto, puede ejecutar trabajos más cortos hasta su finalización, pero a costa del tiempo de respuesta; si en cambio valoras la justicia,

**CONSEJO: LA SUPERPOSICIÓN PERMITE UNA MAYOR UTILIZACIÓN**

Cuando sea posible, superponga las operaciones para maximizar la utilización de los sistemas. La superposición es útil en muchos dominios diferentes, incluso cuando se realiza E/S de disco o se envían mensajes a máquinas remotas; en cualquier caso, iniciar la operación y luego cambiar a otro trabajo es una buena idea y mejora la utilización y eficiencia general del sistema.

El tiempo de respuesta se reduce, pero a costa del tiempo de respuesta. Este tipo de compensación es común en los sistemas; no puedes quedarte con tu pastel y comértelo también<sup>4</sup>.

Hemos desarrollado dos tipos de programadores. El primer tipo (SJF, STCF) optimiza el tiempo de respuesta, pero es malo en cuanto al tiempo de respuesta. El segundo tipo (RR) optimiza el tiempo de respuesta pero es malo para la respuesta. Y todavía tenemos dos supuestos que es necesario relajar: el supuesto 4 (que los trabajos no realizan E/S) y el supuesto 5 (que se conoce el tiempo de ejecución de cada trabajo).

Abordemos esos supuestos a continuación.

## 7.8 Incorporación de E/S

Primero relajaremos el supuesto 4: por supuesto, todos los programas realizan E/S. Imagine un programa que no recibiera ninguna entrada: produciría el mismo resultado cada vez. Imagínese uno sin resultados: es el proverbial árbol que cae en el bosque, sin que nadie lo vea; no importa que se haya ejecutado.

Un programador claramente tiene que tomar una decisión cuando un trabajo inicia una solicitud de E/S, porque el trabajo que se está ejecutando actualmente no utilizará la CPU durante la E/S; está bloqueado esperando que se complete la E/S. Si la E/S se envía a una unidad de disco duro, el proceso podría bloquearse durante unos milisegundos o más, dependiendo de la carga de E/S actual de la unidad. Por lo tanto, el programador probablemente debería programar otro trabajo en la CPU en ese momento.

El planificador también tiene que tomar una decisión cuando se completa la E/S. Cuando eso ocurre, se genera una interrupción y el sistema operativo se ejecuta y mueve el proceso que emitió la E/S del estado bloqueado al estado listo. Por supuesto, incluso podría decidir ejecutar el trabajo en ese momento. ¿Cómo debería tratar el sistema operativo cada trabajo?

Para comprender mejor este problema, supongamos que tenemos dos trabajos, A y B, cada uno de los cuales necesita 50 ms de tiempo de CPU. Sin embargo, hay una diferencia obvia: A se ejecuta durante 10 ms y luego emite una solicitud de E/S (supongamos aquí que cada E/S tarda 10 ms), mientras que B simplemente usa la CPU durante 50 ms y no realiza ninguna E/S. El planificador ejecuta A primero, luego B después (Figura 7.8).

Supongamos que estamos intentando crear un programador STCF. ¿Cómo debería un programador de este tipo tener en cuenta el hecho de que A se divide en 5 subtrabajos de 10 ms?

---

<sup>4</sup>Un dicho que confunde a la gente, porque debería ser "No puedes quedarte con tu pastel y comértelo también" (lo cual es algo obvio, ¿no?). Sorprendentemente, hay una página de Wikipedia sobre este dicho; Aún más sorprendente es que es divertido de leer [W15]. Como dicen en italiano, no se puede avere la botte piena e la moglie ubriaca.

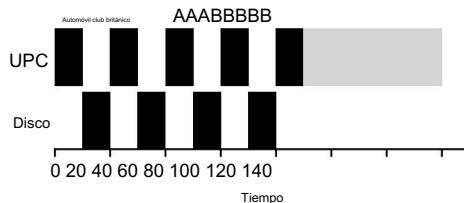


Figura 7.8: Mal uso de los recursos

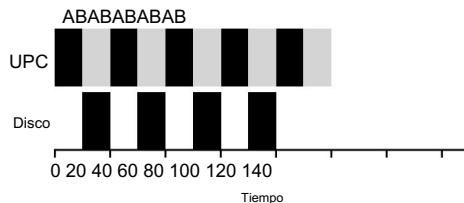


Figura 7.9: La superposición permite un mejor uso de los recursos

¿Mientras que B es solo una demanda de CPU de 50 ms? Claramente, simplemente ejecutar un trabajo y luego el otro sin considerar cómo tener en cuenta la E/S no tiene mucho sentido.

Un enfoque común es tratar cada subtrabajo de 10 ms de A como un trabajo independiente. Por lo tanto, cuando se inicia el sistema, su elección es programar un A de 10 ms o un B de 50 ms. Con STCF, la elección es clara: elegir el más corto, en este caso A. Luego, cuando el primer sub- El trabajo de A se ha completado, solo queda B y comienza a ejecutarse. Luego se envía un nuevo subtrabajo de A, se adelanta a B y se ejecuta durante 10 ms. Hacerlo permite la superposición, con la CPU siendo utilizada por un proceso mientras se espera que se complete la E/S de otro proceso; por lo tanto, el sistema se utiliza mejor (ver Figura 7.9).

Y así vemos cómo un planificador podría incorporar E/S. Al tratar cada ráfaga de CPU como un trabajo, el programador se asegura de que los procesos que son "interactivos" se ejecuten con frecuencia. Mientras esos trabajos interactivos realizan E/S, se ejecutan otros trabajos que requieren un uso intensivo de la CPU, utilizando así mejor el procesador.

## 7.9 No más oráculo

Con un enfoque básico de E/S implementado, llegamos a nuestra suposición final: que el programador conoce la duración de cada trabajo. Como dijimos antes, esta es probablemente la peor suposición que podríamos hacer. De hecho, en un sistema operativo de propósito general (como los que nos interesan), el sistema operativo generalmente sabe muy poco sobre la duración de cada trabajo. Entonces, ¿cómo podemos construir un enfoque que se comporte como SJF/STCF sin ese conocimiento a priori? Además, ¿cómo podemos incorporar algunas de las ideas que hemos visto con el programador RR para que el tiempo de respuesta también sea bastante bueno?

## 7.10 Resumen

Hemos introducido las ideas básicas detrás de la programación y desarrollado dos familias de enfoques. El primero ejecuta el trabajo restante más corto y, por lo tanto, optimiza el tiempo de respuesta; el segundo alterna entre todos los trabajos y así optimiza el tiempo de respuesta. Ambos son malos cuando el otro es bueno, por desgracia, una compensación inherente y común en los sistemas. También hemos visto cómo podríamos incorporar E/S en la imagen, pero aún no hemos resuelto el problema de la incapacidad fundamental del sistema operativo para ver el futuro. En breve veremos cómo superar este problema construyendo un programador que utilice el pasado reciente para predecir el futuro. Este programador se conoce como cola de retroalimentación multinivel y es el tema del próximo capítulo.

## Referencias

[B+79] "El fenómeno del convoy" por M. Blasgen, J. Gray, M. Mitoma, T. Price. ACM Operating Systems Review, 13:2, abril de 1979. Quizás la primera referencia a convoyes, que ocurre tanto en las bases de datos como en el sistema operativo.

[C54] "Asignación de prioridad en problemas de colas de espera" por A. Cobham. Journal of Operations Research, 2:70, páginas 70–76, 1954. El artículo pionero sobre el uso de un enfoque SJF en la programación de la reparación de máquinas.

[K64] "Análisis de un procesador de tiempo compartido" por Leonard Kleinrock. Naval Research Logistics Quarterly, 11:1, páginas 59–73, marzo de 1964. Puede ser la primera referencia al algoritmo de programación por turnos; Sin duda, uno de los primeros análisis de dicho enfoque para programar un sistema de tiempo compartido.

[CK68] "Métodos de programación informática y sus contramedidas" por Edward G. Coffman y Leonard Kleinrock. AFIPS '68 (primavera), abril de 1968. Una excelente introducción temprana y análisis de una serie de disciplinas básicas de programación.

[J91] "El arte del análisis del rendimiento de los sistemas informáticos: técnicas para el diseño, medición, simulación y modelado experimentales" por R. Jain. Interscience, Nueva York, abril de 1991.

El texto estándar sobre medición de sistemas informáticos. Una gran referencia para tu biblioteca, sin duda.

[O45] "Animales en la granja" de George Orwell. Secker y Warburg (Londres), 1945. Un libro alegórico excelente pero deprimente sobre el poder y sus corrupciones. Algunos dicen que es una crítica a Stalin y a la era Stalin anterior a la Segunda Guerra Mundial en la URSS; decimos que es una crítica a los cerdos.

[PV56] "La reparación de máquinas como problema prioritario en la cola de espera" por Thomas E. Phipps Jr., WR Van Voorhis. Operations Research, 4:1, páginas 76–86, febrero de 1956. Trabajo de seguimiento que generaliza el enfoque SJF para la reparación de máquinas a partir del trabajo original de Cobham; También postula la utilidad de un enfoque STCF en ese entorno. Específicamente, "Hay ciertos tipos de trabajos de reparación,... que implican mucho desmantelamiento y recubrimiento del piso con tuercas y pernos, que claramente no deben interrumpirse una vez realizados; en otros casos, sería desaconsejable continuar trabajando en un trabajo largo si uno o más trabajos cortos estuvieran disponibles (p.81)."

[MB91] "El efecto de los cambios de contexto en el rendimiento de la caché" por Jeffrey C. Mogul, Anita Borg. ASPLOS, 1991. Un buen estudio sobre cómo el cambio de contexto puede afectar el rendimiento de la caché; Un problema menor en los sistemas actuales donde los procesadores emiten miles de millones de instrucciones por segundo pero los cambios de contexto aún ocurren en el rango de tiempo de milisegundos.

[W15] "No puedes quedarte con tu pastel y comértelo" por Autores: Desconocido. Wikipedia (a diciembre de 2015). [http://en.wikipedia.org/wiki/No\\_puedes\\_quedarte\\_con\\_tu\\_pastel\\_y\\_comértelo](http://en.wikipedia.org/wiki/No_puedes_quedarte_con_tu_pastel_y_comértelo). - - - - - - - - - -

La mejor parte de esta página es leer todos los modismos similares de otros idiomas. En tamíl no se puede "tener bigote y beber sopa al mismo tiempo".

## Tarea (Simulación)

Este programa, Scheduler.py, le permite ver cómo se desempeñan los diferentes programadores según métricas de programación como el tiempo de respuesta, el tiempo de respuesta y el tiempo total de espera. Consulte el archivo LÉAME para obtener más detalles.

### Preguntas

1. Calcule el tiempo de respuesta y el tiempo de respuesta cuando se ejecutan tres trabajos de longitud 200 con los programadores SJF y FIFO.
2. Ahora haz lo mismo pero con trabajos de diferente duración: 100, 200 y 300.
3. Ahora haz lo mismo, pero también con el planificador RR y un intervalo de tiempo de 1.
4. ¿Para qué tipos de cargas de trabajo SJF ofrece los mismos tiempos de respuesta que FIFO?
5. ¿Para qué tipos de cargas de trabajo y longitudes cuánticas SJF ofrece los mismos tiempos de respuesta que RR?
6. ¿Qué sucede con el tiempo de respuesta de SJF a medida que aumenta la duración de los trabajos? ¿Puedes utilizar el simulador para demostrar la tendencia?
7. ¿Qué sucede con el tiempo de respuesta con RR a medida que aumentan las longitudes cuánticas? ¿Puedes escribir una ecuación que proporcione el tiempo de respuesta en el peor de los casos, dados N trabajos?

## Programación: La cola de comentarios de varios niveles

En este capítulo, abordaremos el problema de desarrollar uno de los enfoques de programación más conocidos, conocido como cola de retroalimentación multinivel (MLFQ). El programador de cola de retroalimentación multinivel (MLFQ) fue descrito por primera vez por Corbato et al. en 1962 [C+62] en un sistema conocido como Compatible Time-Sharing System (CTSS), y este trabajo, junto con trabajos posteriores en Multics, llevó a la ACM a otorgar a Corbato su más alto honor, el Premio Turing. Posteriormente, el programador se ha ido perfeccionando a lo largo de los años para adaptarlo a las implementaciones que encontrará en algunos sistemas modernos.

El problema fundamental que MLFQ intenta abordar es doble. Primero, le gustaría optimizar el tiempo de respuesta, lo cual, como vimos en la nota anterior, se logra ejecutando primero trabajos más cortos; desafortunadamente, el sistema operativo generalmente no sabe cuánto tiempo se ejecutará un trabajo, exactamente el conocimiento que requieren algoritmos como SJF (o STCF). En segundo lugar, a MLFQ le gustaría hacer que un sistema responda a los usuarios interactivos (es decir, usuarios sentados y mirando la pantalla, esperando a que finalice un proceso) y así minimizar el tiempo de respuesta; Desafortunadamente, algoritmos como Round Robin reducen el tiempo de respuesta, pero son terribles en cuanto al tiempo de respuesta. De ahí nuestro problema: dado que en general no sabemos nada sobre un proceso, ¿cómo podemos construir un planificador para lograr estos objetivos? ¿Cómo puede el planificador aprender, mientras el sistema se ejecuta, las características de los trabajos que está ejecutando y así tomar mejores decisiones de programación?

EL CRUZ:

¿ CÓMO PROGRAMAR SIN UN CONOCIMIENTO PERFECTO ?

¿Cómo podemos diseñar un programador que minimice el tiempo de respuesta para trabajos interactivos y al mismo tiempo minimice el tiempo de respuesta sin un conocimiento a priori de la duración del trabajo?

## CONSEJO: APRENDA DE LA

**HISTORIA** La cola de retroalimentación de varios niveles es un excelente ejemplo de un sistema que aprende del pasado para predecir el futuro. Estos enfoques son comunes en los sistemas operativos (y en muchos otros lugares de la informática, incluidos los predictores de ramas de hardware y los algoritmos de almacenamiento en caché). Estos enfoques funcionan cuando los empleos tienen fases de comportamiento y, por tanto, son predecibles; Por supuesto, hay que tener cuidado con tales técnicas, ya que pueden equivocarse fácilmente e impulsar a un sistema a tomar peores decisiones que las que tomaría sin ningún conocimiento.

## 8.1 MLFQ: Reglas básicas

Para construir un programador de este tipo, en este capítulo describiremos los algoritmos básicos detrás de una cola de retroalimentación de múltiples niveles; aunque los detalles específicos de muchos MLFQ implementados difieren [E95], la mayoría de los enfoques son similares.

En nuestro tratamiento, el MLFQ tiene varias colas distintas, a cada una de las cuales se le asigna un nivel de prioridad diferente . En cualquier momento dado, un trabajo que está listo para ejecutarse está en una única cola. MLFQ utiliza prioridades para decidir qué trabajo debe ejecutarse en un momento dado: se elige para ejecutar un trabajo con mayor prioridad (es decir, un trabajo en una cola más alta).

Por supuesto, puede haber más de un trabajo en una cola determinada y, por tanto, tener la misma prioridad. En este caso, simplemente usaremos la programación por turnos entre esos trabajos.

Así, llegamos a las dos primeras reglas básicas para MLFQ:

- Regla 1: Si Prioridad(A) > Prioridad(B), A se ejecuta (B no).
- Regla 2: Si Prioridad(A) = Prioridad(B), A y B se ejecutan en RR.

Por lo tanto, la clave para la programación MLFQ reside en cómo el planificador establece las prioridades. En lugar de dar una prioridad fija a cada trabajo, MLFQ varía la prioridad de un trabajo en función de su comportamiento observado. Si, por ejemplo, un trabajo abandona repetidamente la CPU mientras espera una entrada del teclado, MLFQ mantendrá su prioridad alta, ya que así es como podría comportarse un proceso interactivo. Si, por el contrario, un trabajo utiliza la CPU de forma intensiva durante largos períodos de tiempo, MLFQ reducirá su prioridad. De esta manera, MLFQ intentará aprender sobre los procesos a medida que se ejecutan y así utilizar el historial del trabajo para predecir su comportamiento futuro.

Si tuviéramos que mostrar una imagen de cómo podrían verse las colas en un instante dado, podríamos ver algo como lo siguiente (Figura 8.1).

En la figura, dos trabajos (A y B) tienen el nivel de prioridad más alto, mientras que el trabajo C está en el medio y el trabajo D tiene la prioridad más baja. Dado nuestro conocimiento actual de cómo funciona MLFQ, el programador simplemente alternaría intervalos de tiempo entre A y B porque son los trabajos de mayor prioridad en el sistema; Los pobres empleos C y D ni siquiera llegarían a postularse: ¡una barbaridad!

Por supuesto, mostrar simplemente una instantánea estática de algunas colas no da realmente una idea de cómo funciona MLFQ. Lo que necesitamos es comprender

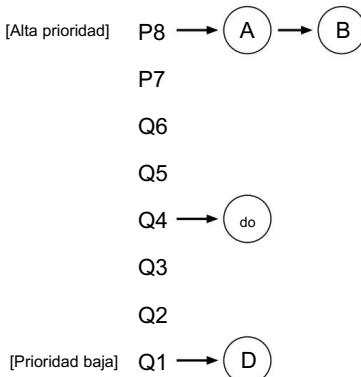


Figura 8.1: Ejemplo de MLFQ

comprender cómo la prioridad laboral cambia con el tiempo. Y eso, para sorpresa sólo de aquellos que leen un capítulo de este libro por primera vez, es exactamente lo que haremos a continuación.

## 8.2 Intento n.º 1: Cómo cambiar la prioridad

Ahora debemos decidir cómo MLFQ va a cambiar el nivel de prioridad de un trabajo (y por lo tanto en qué cola se encuentra) durante la vida útil de un trabajo. Para hacer esto, debemos tener en cuenta nuestra carga de trabajo: una combinación de trabajos interactivos que duran poco tiempo (y que con frecuencia pueden renunciar a la CPU) y algunos trabajos de ejecución más larga “dependientes de la CPU” que necesitan mucho tiempo de CPU pero donde el tiempo de respuesta no es importante. Aquí está nuestro primer intento de un algoritmo de ajuste de prioridad:

- Regla 3: Cuando un trabajo ingresa al sistema, se coloca en la prioridad más alta (la cola más alta).
- Regla 4a: Si un trabajo utiliza un intervalo de tiempo completo mientras se ejecuta, su prioridad se reduce (es decir, baja una cola).
- Regla 4b: si un trabajo abandona la CPU antes de que finalice el intervalo de tiempo, permanece al mismo nivel de prioridad.

### Ejemplo 1: un único trabajo de larga duración Veamos

algunos ejemplos. Primero, veremos qué sucede cuando ha habido un trabajo en ejecución durante mucho tiempo en el sistema. La Figura 8.2 muestra lo que sucede con este trabajo a lo largo del tiempo en un planificador de tres colas.

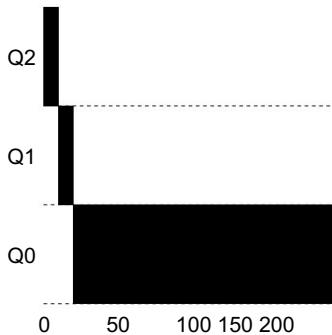


Figura 8.2: Trabajo de larga duración a lo largo del tiempo

Como puedes ver en el ejemplo, el trabajo entra con la máxima prioridad.

(Q2). Después de un único intervalo de tiempo de 10 ms, el programador reduce el tiempo del trabajo. prioridad por uno y, por lo tanto, el trabajo está en Q1. Despues de estar en la Q1 por un tiempo segmento, el trabajo finalmente se reduce a la prioridad más baja en el sistema (Q0), donde permanece. Bastante simple, ¿no?

### Ejemplo 2: Llegó un trabajo breve

Ahora veamos un ejemplo más complicado y, con suerte, veamos cómo MLFQ intenta aproximarse a SJF. En este ejemplo, hay dos trabajos: A, que es un trabajo de larga duración que consume mucha CPU, y B, que es un trabajo de corta duración. trabajo interactivo. Supongamos que A ha estado funcionando durante algún tiempo y luego llega B. ¿Qué pasará? ¿MLFQ se aproximará a SJF para B?

La Figura 8.3 muestra los resultados de este escenario. A (que se muestra en negro) se está ejecutando en la cola de menor prioridad (al igual que cualquier trabajo de larga duración que requiera un uso intensivo de la CPU); B (que se muestra en gris) llega al momento T = 100 y, por lo tanto, es

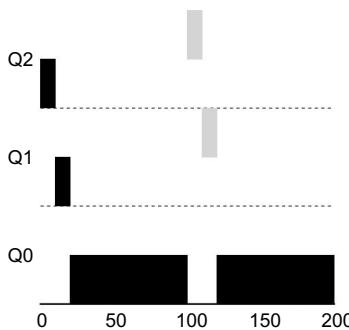


Figura 8.3: Llegó un trabajo interactivo

## PROGRAMACIÓN:

## LA COLA DE COMENTARIOS MULTINIVEL

5

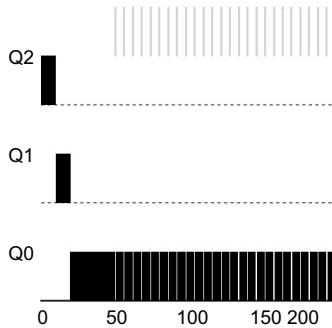


Figura 8.4: Una carga de trabajo mixta con uso intensivo de E/S y CPU insertada en la cola

más alta; como su tiempo de ejecución es corto (sólo 20 ms), B se completa antes de llegar a la cola inferior, en dos intervalos de tiempo; luego A continúa ejecutándose (con baja prioridad).

A partir de este ejemplo, se puede entender uno de los objetivos principales del algoritmo: debido a que no sabe si un trabajo será corto o de larga duración, primero asume que podría ser un trabajo corto, dando así el trabajo es de alta prioridad. Si realmente es un trabajo corto, se ejecutará rápidamente y se completará; si no es un trabajo corto, avanzará lentamente en las colas y, por lo tanto, pronto demostrará ser un proceso de larga duración, más parecido a un lote.

De esta manera, MLFQ se aproxima a SJF.

### Ejemplo 3: ¿Qué pasa con las E/S?

Veamos ahora un ejemplo con algunas E/S. Como establece la Regla 4b anteriormente, si un proceso abandona el procesador antes de agotar su intervalo de tiempo, lo mantenemos en el mismo nivel de prioridad. La intención de esta regla es simple: si un trabajo interactivo, por ejemplo, realiza muchas E/S (por ejemplo, esperando la entrada del usuario desde el teclado o el mouse), abandonará la CPU antes de que se complete su intervalo de tiempo; en tal caso, no queremos penalizar el trabajo y simplemente mantenerlo al mismo nivel.

La Figura 8.4 muestra un ejemplo de cómo funciona esto, con un trabajo interactivo B (que se muestra en gris) que necesita la CPU solo durante 1 ms antes de realizar una E/S que compite por la CPU con un trabajo por lotes A de larga duración (que se muestra en negro). El enfoque MLFQ mantiene a B con la máxima prioridad porque B sigue liberando la CPU; Si B es un trabajo interactivo, MLFQ logra aún más su objetivo de ejecutar trabajos interactivos rápidamente.

Problemas con nuestro MLFQ actual Por lo tanto,

tenemos un MLFQ básico. Parece hacer un trabajo bastante bueno, compartiendo la CPU de manera justa entre trabajos de larga duración y permitiendo que trabajos interactivos cortos o con uso intensivo de E/S se ejecuten rápidamente. Desafortunadamente, el enfoque que hemos desarrollado hasta ahora contiene graves fallas. ¿Se te ocurre alguno?

(Aquí es donde haces una pausa y piensas tan tortuosamente como puedas)

## PROGRAMACIÓN:

6

## LA COLA DE COMENTARIOS MULTINIVEL

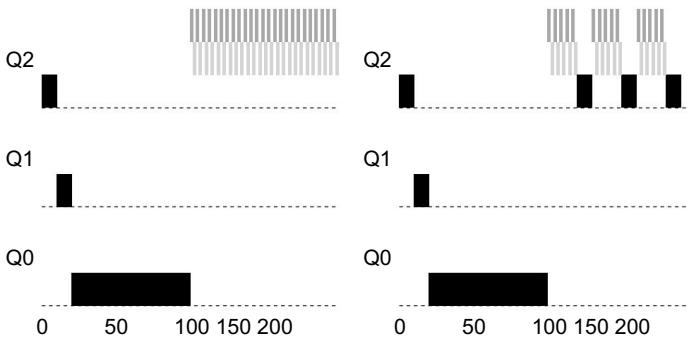


Figura 8.5: Sin (izquierda) y con (derecha) refuerzo de prioridad

Primero, está el problema de la inanición: si hay “demasiados” trabajos interactivos en el sistema, se combinarán para consumir todo el tiempo de la CPU. y, por lo tanto, los trabajos de larga duración nunca recibirán tiempo de CPU (se mueren de hambre). Nos gustaría lograr algunos avances en estos trabajos incluso en este escenario.

En segundo lugar, un usuario inteligente podría reescribir su programa para engañar al programador. Jugar con el programador generalmente se refiere a la idea de hacer algo furtivo para engañar al programador y que le dé más de lo que le corresponde. La participación del recurso. El algoritmo que hemos descrito es susceptible a el siguiente ataque: antes de que finalice el intervalo de tiempo, emita una operación de E/S (a algún archivo que no te interesa) y así renunciar a la CPU; haciéndolo le permite permanecer en la misma cola y, por lo tanto, ganar un mayor porcentaje de tiempo de CPU. Cuando se hace correctamente (por ejemplo, ejecutándose durante el 99% de un intervalo de tiempo) antes de renunciar a la CPU, un trabajo podría casi monopolizar la CPU.

Finalmente, un programa puede cambiar su comportamiento con el tiempo; lo que estaba ligado a la CPU puede pasar a una fase de interactividad. Con nuestro enfoque actual, un trabajo así no tendría suerte y no sería tratado como el otro. trabajos interactivos en el sistema.

## CONSEJO: LA PROGRAMACIÓN DEBE ESTAR PROTEGIDA DE ATAQUES

Se podría pensar que una política de programación, ya sea dentro del propio sistema operativo (como se analiza en este documento), o en un contexto más amplio (por ejemplo, en el manejo de solicitudes de E/S de un sistema de almacenamiento distribuido [Y+18]), no es un problema de seguridad , pero en muchos casos, es exactamente eso. Consideremos el moderno centro de datos, en el que usuarios de todo el mundo comparten CPU, memorias, redes y sistemas de almacenamiento; Sin cuidado en el diseño y aplicación de políticas, un solo usuario puede dañar negativamente a otros y ganar. ventaja para sí mismo. Por lo tanto, la política de programación forma una parte importante de la seguridad de un sistema y debe construirse cuidadosamente.

PROGRAMACIÓN:  
LA COLA DE COMENTARIOS MULTINIVEL

7

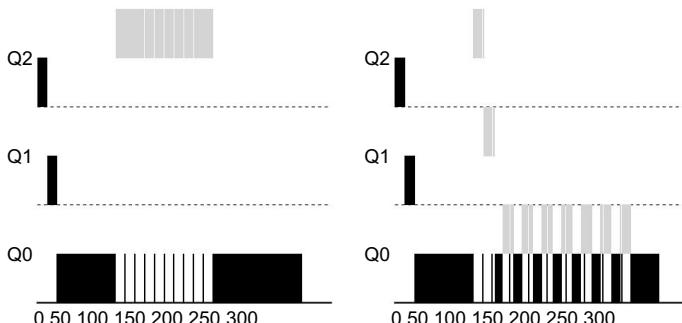


Figura 8.6: Sin (izquierda) y con (derecha) tolerancia al juego

### 8.3 Intento n.º 2: el impulso de prioridad

Intentemos cambiar las reglas y ver si podemos evitar el problema del hambre. ¿Qué podríamos hacer para garantizar que los trabajos vinculados a la CPU progresen algo (aunque no sea mucho?).

La idea simple aquí es aumentar periódicamente la prioridad de todos los trabajos en el sistema. Hay muchas maneras de lograr esto, pero hagamos algo simple: colóquelos todos en la cola superior; por tanto, una nueva regla:

- Regla 5: Después de un período de tiempo  $S$ , mueva todos los trabajos del sistema a la cola superior.

Nuestra nueva regla resuelve dos problemas a la vez. En primer lugar, se garantiza que los procesos no morirán de hambre: al permanecer en la cola superior, un trabajo compartirá la CPU con otros trabajos de alta prioridad en forma circular y, por lo tanto, eventualmente recibirá el servicio. En segundo lugar, si un trabajo vinculado a la CPU se ha vuelto interactivo, el programador lo trata adecuadamente una vez que ha recibido el aumento de prioridad.

Veamos un ejemplo. En este escenario, simplemente mostramos el comportamiento de un trabajo de ejecución prolongada cuando compite por la CPU con dos trabajos interactivos de ejecución corta. En la Figura 8.5 (página 6) se muestran dos gráficos. En la izquierda, no hay ningún impulso prioritario y, por lo tanto, el trabajo de larga duración se queda sin trabajo una vez que llegan los dos trabajos cortos; a la derecha, hay un aumento de prioridad cada 50 ms (que probablemente sea un valor demasiado pequeño, pero se usa aquí como ejemplo) y, por lo tanto, al menos garantizamos que el trabajo de larga duración progresará un poco y se impulsará a la prioridad más alta cada 50 ms y así llegar a ejecutarse periódicamente.

Por supuesto, la adición del período de tiempo  $S$  lleva a la pregunta obvia: ¿a qué valor debería fijarse  $S$ ? John Ousterhout, un reconocido investigador de sistemas [O11], solía llamar a estos valores en los sistemas constantes vudú, porque parecían requerir alguna forma de magia negra para establecerlos correctamente. Desafortunadamente,  $S$  tiene ese sabor. Si se fija demasiado alto, los empleos de larga duración podrían morir de hambre; demasiado bajo y es posible que los trabajos interactivos no obtengan una proporción adecuada de la CPU.

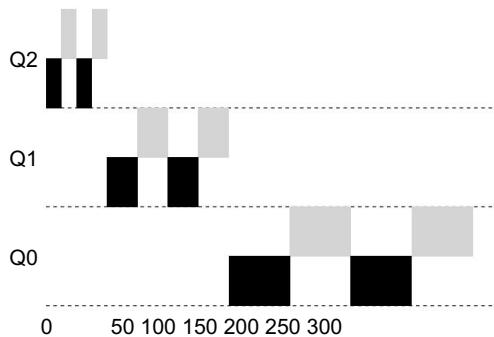


Figura 8.7: Prioridad más baja, cuantos más largos

#### 8.4 Intento #3: Mejor Contabilidad Ahora tenemos un

problema más que resolver: ¿cómo evitar que nuestro programador juegue? El verdadero culpable aquí, como habrás adivinado, son las Reglas 4a y 4b, que permiten que un trabajo conserve su prioridad al renunciar a la CPU antes de que expire el intervalo de tiempo. Entonces, ¿qué debemos hacer?

La solución aquí es realizar una mejor contabilidad del tiempo de CPU en cada nivel del MLFQ. En lugar de olvidar cuánto tiempo utilizó un proceso en un nivel determinado, el planificador debe realizar un seguimiento; Una vez que un proceso ha utilizado su asignación, se degrada a la siguiente cola de prioridad. No importa si utiliza el intervalo de tiempo en una ráfaga larga o en muchas pequeñas.

Por lo tanto, reescribimos las Reglas 4a y 4b a la siguiente regla única:

- Regla 4: Una vez que un trabajo agota su asignación de tiempo en un nivel determinado (independientemente de cuántas veces haya renunciado a la CPU), su prioridad se reduce (es decir, baja una cola).

Veamos un ejemplo. La Figura 8.6 (página 7) muestra lo que sucede cuando una carga de trabajo intenta engañar al planificador con las antiguas Reglas 4a y 4b (a la izquierda), así como con la nueva Regla anti-juego 4. Sin ninguna protección contra el juego, un proceso puede emitir una E/S justo antes de que finalice un intervalo de tiempo y, por lo tanto, domina el tiempo de CPU. Con tales protecciones implementadas, independientemente del comportamiento de E/S del proceso, este avanza lentamente en las colas y, por lo tanto, no puede obtener una parte injusta de la CPU.

#### 8.5 Ajuste de MLFQ y otros problemas Surgen algunos

otros problemas con la programación de MLFQ. Una gran pregunta es cómo parametrizar dicho programador. Por ejemplo, ¿cuántas colas debería haber? ¿Qué tamaño debe tener el intervalo de tiempo por cola? ¿Con qué frecuencia se deben aumentar las prioridades para evitar el hambre y tener en cuenta los cambios de comportamiento? No hay respuestas fáciles a estas preguntas y, por lo tanto, sólo un poco de experiencia con las cargas de trabajo y el posterior ajuste del programador conducirán a un equilibrio satisfactorio.

## PROGRAMACIÓN:

## LA COLA DE COMENTARIOS MULTINIVEL

9

## CONSEJO: EVITE LAS CONSTANTES VOO-DOO (LEY DE OUSTERHOUT )

Evitar las constantes vudú es una buena idea siempre que sea posible. Desafortunadamente, como en el ejemplo anterior, a menudo resulta difícil. Se podría intentar hacer que el sistema aprenda un buen valor, pero eso tampoco es sencillo.

El resultado frecuente: un archivo de configuración lleno de valores de parámetros predeterminados que un administrador experimentado puede modificar cuando algo no funciona correctamente. Como puede imaginar, estos a menudo no se modifican y, por lo tanto, nos queda la esperanza de que los valores predeterminados funcionen bien en el campo. Este consejo lo trajo nuestro antiguo profesor de SO, John Ousterhout, y por eso lo llamamos Ley de Ousterhout.

Por ejemplo, la mayoría de las variantes de MLFQ permiten variar la duración del intervalo de tiempo en diferentes colas. A las colas de alta prioridad generalmente se les asignan períodos de tiempo cortos; después de todo, se componen de trabajos interactivos y, por lo tanto, tiene sentido alternar rápidamente entre ellos (por ejemplo, 10 milisegundos o menos). Por el contrario, las colas de baja prioridad contienen trabajos de larga duración que están vinculados a la CPU; por lo tanto, intervalos de tiempo más largos funcionan bien (p. ej., cientos de ms). La Figura 8.7 (página 8) muestra un ejemplo en el que dos trabajos se ejecutan durante 20 ms en la cola más alta (con un intervalo de tiempo de 10 ms), 40 ms en el medio (intervalo de tiempo de 20 ms) y con un intervalo de tiempo de 40 ms. intervalo de tiempo en el más bajo.

La implementación de Solaris MLFQ (la clase de programación de tiempo compartido o TS) es particularmente fácil de configurar; proporciona un conjunto de tablas que determinan exactamente cómo se modifica la prioridad de un proceso a lo largo de su vida útil, cuánto dura cada intervalo de tiempo y con qué frecuencia se debe aumentar la prioridad de un trabajo [AD00]; un administrador puede modificar esta tabla para que el programador se comporte de diferentes maneras. Los valores predeterminados para la tabla son 60 colas, con longitudes de intervalo de tiempo que aumentan lentamente desde 20 milisegundos (prioridad más alta) hasta unos pocos cientos de milisegundos (la más baja), y las prioridades aumentan aproximadamente cada 1 segundo aproximadamente.

Otros programadores MLFQ no utilizan una tabla ni las reglas exactas descritas en este capítulo; más bien ajustan las prioridades utilizando fórmulas matemáticas. Por ejemplo, el programador de FreeBSD (versión 4.3) usa una fórmula para calcular el nivel de prioridad actual de un trabajo, basándose en cuánta CPU ha usado el proceso [LM+89]; Además, el uso disminuye con el tiempo, proporcionando el aumento de prioridad deseado de una manera diferente a la descrita en este documento. Consulte el artículo de Epema para obtener una excelente descripción general de dichos algoritmos de uso de desintegración y sus propiedades [E95].

Finalmente, muchos programadores tienen algunas otras características que usted podría encontrar. Por ejemplo, algunos programadores reservan los niveles de prioridad más altos para el trabajo del sistema operativo; por lo tanto, los trabajos de usuario típicos nunca pueden obtener los niveles más altos de prioridad en el sistema. Algunos sistemas también permiten algunos consejos al usuario para ayudar a establecer prioridades; por ejemplo, al utilizar la utilidad de línea de comandos nice, puede aumentar o disminuir la prioridad de un trabajo (un poco) y así aumentar o disminuir sus posibilidades de ejecutarse en un momento dado.

Consulte la página de manual para obtener más información.

## CONSEJO: UTILICE LOS CONSEJOS CUANDO SEA POSIBLE

Como el sistema operativo rara vez sabe qué es lo mejor para todos y cada uno de los procesos del sistema, suele ser útil proporcionar interfaces que permitan a los usuarios o administradores proporcionar algunas sugerencias al sistema operativo. A menudo llamamos a estas sugerencias consejos, ya que el sistema operativo no necesariamente tiene que prestarles atención, sino que puede tener en cuenta los consejos para tomar una mejor decisión.

Estas sugerencias son útiles en muchas partes del sistema operativo, incluido el programador (p. ej., con nice), el administrador de memoria (p. ej., madvise) y el sistema de archivos (p. ej., captación previa informada y almacenamiento en caché [P+95]).

## 8.6 MLFQ: Resumen

Hemos descrito un enfoque de programación conocido como cola de comentarios multinivel (MLFQ). Con suerte, ahora podrá ver por qué se llama así: tiene múltiples niveles de colas y utiliza comentarios para determinar la prioridad de un trabajo determinado. La historia es su guía: preste atención a cómo se comportan los trabajos a lo largo del tiempo y trátelos en consecuencia.

El conjunto refinado de reglas MLFQ, repartidas a lo largo del capítulo, se reproduce aquí para su placer visual:

- Regla 1: Si Prioridad(A) > Prioridad(B), A se ejecuta (B no). • Regla 2: Si Prioridad(A) = Prioridad(B), A y B se ejecutan en modo round-robin utilizando el segmento de tiempo (longitud cuántica) de la cola dada. • Regla 3: Cuando un trabajo ingresa al sistema, se coloca en la prioridad más alta (la cola más alta). • Regla 4: Una vez que un trabajo agota su asignación de tiempo en un nivel determinado (independientemente de cuántas veces haya renunciado a la CPU), su prioridad se reduce (es decir, baja una cola). • Regla 5: Despues de un período de tiempo S, mueva todos los trabajos del sistema a la cola superior.

MLFQ es interesante por la siguiente razón: en lugar de exigir un conocimiento a priori de la naturaleza de un trabajo, observa la ejecución de un trabajo y lo prioriza en consecuencia. De esta manera, logra lograr lo mejor de ambos mundos: puede ofrecer un rendimiento general excelente (similar a SJF/STCF) para trabajos interactivos de corta duración, y es justo y avanza para cargas de trabajo de larga duración que requieren un uso intensivo de la CPU. Por esta razón, muchos sistemas, incluidos los derivados de BSD UNIX [LM+89, B86], Solaris [M06] y Windows NT y los sistemas operativos Windows posteriores [CS97] utilizan una forma de MLFQ como programador base.

## PROGRAMACIÓN:

## LA COLA DE COMENTARIOS MULTINIVEL

11

## Referencias

[AD00] "Programación de colas de comentarios multinivel en Solaris" por Andrea Arpaci-Dusseau. Disponible: <http://www.ostep.org/Citations/notes-solaris.pdf>. Un excelente conjunto breve de notas de uno de los autores sobre los detalles del programador Solaris. Vale, probablemente seamos parciales en esta descripción, pero las notas son bastante buenas.

[B86] "El diseño del sistema operativo UNIX" de MJ Bach. Prentice-Hall, 1986. Uno de los libros clásicos sobre cómo se construye un sistema operativo UNIX real; una lectura obligada para los hackers del kernel.

[C+62] "Un sistema experimental de tiempo compartido" por FJ Corbató, MM Daggett, RC Daley.

IFIPS 1962. Un poco difícil de leer, pero es la fuente de muchas de las primeras ideas en la programación de retroalimentación multinivel. Gran parte de esto se destinó posteriormente a Multics, que se podría argumentar que fue el sistema operativo más influyente de todos los tiempos.

[CS97] "Dentro de Windows NT" por Helen Custer y David A. Solomon. Prensa de Microsoft, 1997.

El libro de NT, si desea aprender sobre algo que no sea UNIX. Por supuesto, ¿por qué lo harías? Bien, estamos bromeando; Es posible que algún día trabajes para Microsoft.

[E95] "Un análisis de la programación del uso de decadencia en multiprocesadores" por DHJ Epema. SIG-MÉTRICAS '95. Un buen artículo sobre el estado del arte de la programación de mediados de la década de 1990, que incluye una buena descripción general del enfoque básico detrás de los programadores de uso de decaimiento.

[LM+89] "El diseño y la implementación del sistema operativo UNIX 4.3BSD" por SJ Lef-fler, MK McKusick, MJ Karels, JS Quarterman. Addison-Wesley, 1989. Otro clásico de los sistemas operativos, escrito por cuatro de las principales personas detrás de BSD. Las versiones posteriores de este libro, aunque más actualizadas, no igualan la belleza de éste.

[M06] "Solaris Internals: Solaris 10 y OpenSolaris Kernel Architecture" por Richard Mc-Dougall. Prentice-Hall, 2006. Un buen libro sobre Solaris y cómo funciona.

[O11] "Página de inicio de John Ousterhout" por John Ousterhout. [www.stanford.edu/~ouster/](http://www.stanford.edu/~ouster/).

La página de inicio del famoso profesor Ousterhout. Los dos coautores de este libro tuvieron el placer de aprender sistemas operativos de posgrado de Ousterhout mientras estaban en la escuela de posgrado; de hecho, aquí es donde los dos coautores se conocieron, lo que finalmente los llevó al matrimonio, a tener hijos e incluso a este libro. Por lo tanto, realmente puedes culpar a Ousterhout por todo este lío en el que te encuentras.

[P+95] "Captación previa y almacenamiento en caché informados" por RH Patterson, GA Gibson, E. Gingting, D. Stodolsky, J. Zelenka. SOSP '95, Copper Mountain, Colorado, octubre de 1995. Un artículo divertido sobre algunas ideas muy interesantes en sistemas de archivos, incluyendo cómo las aplicaciones pueden dar consejos al sistema operativo sobre a qué archivos está accediendo y cómo planea acceder a ellos.

[Y+18] "Análisis de programabilidad de principios para sistemas de almacenamiento distribuido utilizando modelos de arquitectura de subprocesos" por Suli Yang, Jing Liu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. OSDI '18, San Diego, California. Un trabajo reciente de nuestro grupo que demuestra la dificultad de programar solicitudes de E/S dentro de sistemas de almacenamiento distribuidos modernos como Hive/HDFS, Cassandra, MongoDB y Riak. Sin cuidado, un solo usuario podría monopolizar la información del sistema.  
fuentes.

## Tarea (Simulación)

Este programa, mlfq.py, le permite ver cómo funciona el programador MLFQ. Se comporta lo que se presenta en este capítulo. Consulte el archivo LÉAME para obtener más detalles.

### Preguntas

1. Ejecutar algunos problemas generados aleatoriamente con sólo dos trabajos y dos colas; calcule el seguimiento de ejecución de MLFQ para cada uno. Haga su vida más fácil limitando la duración de cada trabajo y desactivando las E/S.
2. ¿Cómo ejecutarías el programador para reproducir cada uno de los ejemplos del capítulo?
3. ¿Cómo configuraría los parámetros del programador para que se comporten como un programador por turnos?
4. Cree una carga de trabajo con dos trabajos y parámetros del programador para que un trabajo aproveche las reglas 4a y 4b más antiguas (activadas con el indicador -S) para jugar con el programador y obtener el 99 % de la CPU durante un intervalo de tiempo particular.
5. Dado un sistema con una longitud cuántica de 10 ms en su cola más alta, ¿con qué frecuencia tendría que impulsar los trabajos nuevamente al nivel de prioridad más alto (con el indicador -B) para garantizar que un único sistema de larga ejecución (y ¿El trabajo potencialmente hambriento obtiene al menos el 5% de la CPU?)
6. Una pregunta que surge en la programación es en qué extremo de una cola agregar un trabajo que acaba de finalizar la E/S; la bandera -l cambia este comportamiento para este simulador de programación. Experimente con algunas cargas de trabajo y vea si puede ver el efecto de esta bandera.

## La abstracción: espacios de direcciones

Al principio, construir sistemas informáticos era fácil. ¿Por qué, preguntas? Porque los usuarios no esperaban mucho. Son esos malditos usuarios con sus expectativas de “facilidad de uso”, “alto rendimiento”, “confiabilidad”, etc., los que realmente han provocado todos estos dolores de cabeza. La próxima vez que conozca a uno de esos usuarios de computadoras, agradézcale por todos los problemas que ha causado.

### 13.1 Sistemas tempranos

Desde la perspectiva de la memoria, las primeras máquinas no proporcionaban mucha abstracción a los usuarios. Básicamente, la memoria física de la máquina se parecía a lo que se ve en la Figura 13.1 (página 2).

El sistema operativo era un conjunto de rutinas (una biblioteca, en realidad) que se encontraban en la memoria (comenzando en la dirección física 0 en este ejemplo), y habría un programa en ejecución (un proceso) que actualmente se encontraba en la memoria física (comenzando en la dirección física 64k en este ejemplo) y usó el resto de la memoria. Había pocas ilusiones aquí y el usuario no esperaba mucho del sistema operativo. La vida era fácil para los desarrolladores de sistemas operativos en aquellos días, ¿no?

### 13.2 Multiprogramación y tiempo compartido

Después de un tiempo, como las máquinas eran caras, la gente empezó a compartir las de forma más eficaz. Así nació la era de la multiprogramación [DV66], en la que múltiples procesos estaban listos para ejecutarse en un momento dado, y el sistema operativo cambiaba entre ellos, por ejemplo cuando uno decidía realizar una E/S. Al hacerlo, aumentó la utilización efectiva de la CPU. Estos aumentos de eficiencia eran particularmente importantes en aquellos días en los que cada máquina costaba cientos de miles o incluso millones de dólares (¡y uno pensaba que su Mac era cara!).

Sin embargo, muy pronto la gente empezó a exigir más de las máquinas y nació la era del tiempo compartido [S59, L60, M62, M83]. Específicamente, muchos se dieron cuenta de las limitaciones de la computación por lotes, particularmente entre los propios programadores [CV65], que estaban cansados de largas (y por lo tanto ineficaces)

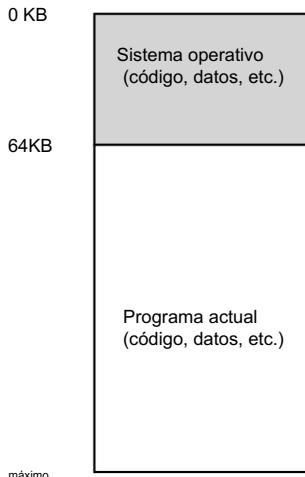


Figura 13.1: Sistemas operativos: los primeros días

tivo) ciclos de depuración del programa. La noción de interactividad se volvió importante, ya que muchos usuarios podían estar usando simultáneamente una máquina, cada uno esperando (o deseando) una respuesta oportuna de sus tareas en ejecución en ese momento.

Una forma de implementar el tiempo compartido sería ejecutar un proceso por un corto tiempo, dándole acceso completo a toda la memoria (Figura 13.1), luego detenerlo, guardar todo su estado en algún tipo de disco (incluidos todos los miembros físicos). ory), cargar el estado de algún otro proceso, ejecutarlo por un tiempo, y así implementar algún tipo de intercambio crudo de la máquina [M+63].

Desafortunadamente, este enfoque tiene un gran problema: es demasiado lento, especialmente a medida que crece la memoria. Si bien guardar y restaurar el estado a nivel de registro (la PC, registros de propósito general, etc.) es relativamente rápido, guardar todo el contenido de la memoria en el disco tiene un rendimiento brutalmente deficiente. Por lo tanto, lo que preferiríamos hacer es dejar los procesos en la memoria mientras cambiamos entre ellos, permitiendo que el sistema operativo implemente el tiempo compartido de manera eficiente (como se muestra en la Figura 13.2, página 3).

En el diagrama, hay tres procesos (A, B y C) y cada uno de ellos tiene una pequeña parte de la memoria física de 512 KB reservada para ellos. Suponiendo una sola CPU, el sistema operativo elige ejecutar uno de los procesos (digamos A), mientras que los demás (B y C) se encuentran en la cola listos esperando para ejecutarse.

A medida que el tiempo compartido se hizo más popular, probablemente puedas adivinar que se impusieron nuevas demandas al sistema operativo. En particular, permitir que múltiples programas residan simultáneamente en la memoria hace que la protección sea una cuestión importante; no desea que un proceso pueda leer, o peor aún, escribir la memoria de otro proceso.

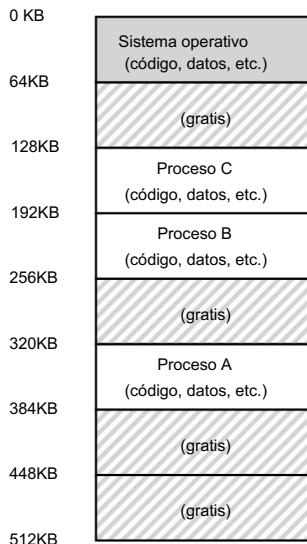


Figura 13.2: Tres procesos: compartir memoria

### 13.3 El espacio de direcciones

Sin embargo, debemos tener en cuenta a esos molestos usuarios, y para ello es necesario que el sistema operativo cree una abstracción de memoria física fácil de usar.

A esta abstracción la llamamos espacio de direcciones y es la vista que tiene el programa en ejecución de la memoria en el sistema. Comprender esta abstracción fundamental de la memoria del sistema operativo es clave para comprender cómo se virtualiza la memoria.

El espacio de direcciones de un proceso contiene todo el estado de la memoria del programa en ejecución. Por ejemplo, el código del programa (las instrucciones) tiene que vivir en algún lugar de la memoria y, por lo tanto, está en el espacio de direcciones. El programa, mientras se ejecuta, utiliza una pila para realizar un seguimiento de dónde se encuentra en la cadena de llamadas a funciones, así como para asignar variables locales y pasar parámetros y valores de retorno hacia y desde las rutinas. Finalmente, el montón se utiliza para memoria administrada por el usuario y asignada dinámicamente, como la que podría recibir de una llamada a malloc() en C o new en un lenguaje orientado a objetos como C++ o Java. Por supuesto, también hay otras cosas allí (por ejemplo, variables inicializadas estáticamente), pero por ahora supongamos esos tres componentes: código, pila y montón.

En el ejemplo de la Figura 13.3 (página 4), tenemos un pequeño espacio de direcciones (sólo 16 KB)<sup>1</sup>. El código del programa se encuentra en la parte superior del espacio de direcciones.

<sup>1</sup>A menudo usaremos pequeños ejemplos como este porque (a) es complicado representar un sistema de 32 bits espacio de direcciones y (b) las matemáticas son más difíciles. Nos gustan las matemáticas simples.

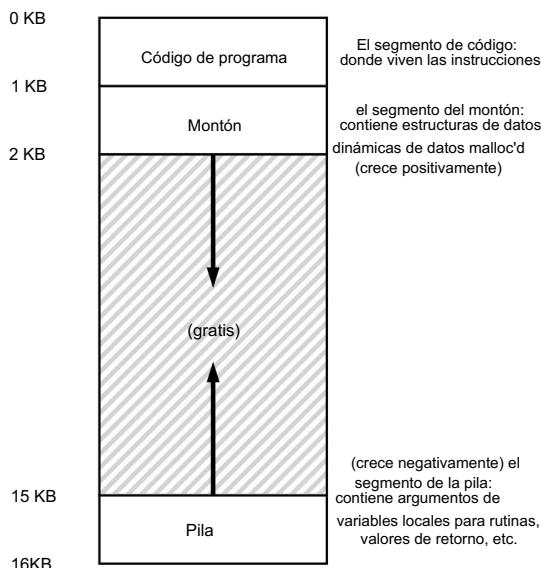


Figura 13.3: Un ejemplo de espacio de direcciones

(comenzando en 0 en este ejemplo y está empaquetado en el primer 1K del espacio de direcciones). El código es estático (y por lo tanto fácil de colocar en la memoria), por lo que podemos colocarlo en la parte superior del espacio de direcciones y saber que no necesitará más espacio mientras se ejecuta el programa.

A continuación, tenemos las dos regiones del espacio de direcciones que pueden crecer (y reducirse) mientras se ejecuta el programa. Esos son el montón (en la parte superior) y la pila (en la parte inferior). Los colocamos así porque cada uno desea poder crecer, y al colocarlos en extremos opuestos del espacio de direcciones, podemos permitir dicho crecimiento: simplemente tienen que crecer en direcciones opuestas. Por lo tanto, el montón comienza justo después del código (en 1 KB) y crece hacia abajo (por ejemplo, cuando un usuario solicita más memoria a través de malloc()); la pila comienza en 16 KB y crece hacia arriba (por ejemplo, cuando un usuario realiza una llamada de procedimiento). Sin embargo, esta ubicación de pila y montón es sólo una convención; puede organizar el espacio de direcciones de una manera diferente si lo desea (como veremos más adelante, cuando coexisten varios subprocessos en un espacio de direcciones, lamentablemente ya no funciona una buena manera de dividir el espacio de direcciones de esta manera).

Por supuesto, cuando describimos el espacio de direcciones, lo que estamos describiendo es la abstracción que el sistema operativo proporciona al programa en ejecución. El programa realmente no está en la memoria en las direcciones físicas de 0 a 16 KB; más bien se carga en alguna dirección física arbitraria. Examine los procesos A, B y C en la figura 13.2; allí puedes ver cómo cada proceso se carga en la memoria en una dirección diferente. Y de ahí el problema:

### EL CRUX: CÓMO VIRTUALIZAR LA MEMORIA

¿Cómo puede el sistema operativo construir esta abstracción de un espacio de direcciones privado y potencialmente grande para múltiples procesos en ejecución (todos compartiendo memoria) sobre una única memoria física?

Cuando el sistema operativo hace esto, decimos que está virtualizando la memoria, porque el programa en ejecución cree que está cargado en la memoria en una dirección particular (digamos 0) y tiene un espacio de direcciones potencialmente muy grande (digamos 32 bits o 64 bits); la realidad es bastante diferente.

Cuando, por ejemplo, el proceso A en la Figura 13.2 intenta realizar una carga en la dirección 0 (a la que llamaremos dirección virtual), de alguna manera el sistema operativo, junto con algún soporte de hardware, tendrá que asegurarse de que la carga en realidad no lo haga. vaya a la dirección física 0 sino a la dirección física 320 KB (donde A está cargado en la memoria). Ésta es la clave para la virtualización de la memoria, que es la base de todos los sistemas informáticos modernos del mundo.

## 13.4 Metas

Así llegamos al trabajo del sistema operativo en este conjunto de notas: virtualizar la memoria. Sin embargo, el sistema operativo no sólo virtualizará la memoria; lo hará con estilo. Para asegurarnos de que el sistema operativo lo haga, necesitamos algunos objetivos que nos guíen. Hemos visto estos objetivos antes (piense en la Introducción) y los veremos nuevamente, pero ciertamente vale la pena repetirlos.

Uno de los principales objetivos de un sistema de memoria virtual (VM) es la transparencia<sup>2</sup> . El sistema operativo debe implementar la memoria virtual de una manera que sea invisible para el programa en ejecución. Por tanto, el programa no debería ser consciente del hecho de que la memoria está virtualizada; más bien, el programa se comporta como si tuviera su propia memoria física privada. Detrás de escena, el sistema operativo (y el hardware) hace todo el trabajo para multiplexar la memoria entre muchos trabajos diferentes y, por lo tanto, implementa la ilusión.

Otro objetivo de VM es la eficiencia. El sistema operativo debe esforzarse por hacer que la virtualización sea lo más eficiente posible, tanto en términos de tiempo (es decir, no hacer que los programas se ejecuten mucho más lentamente) como de espacio (es decir, no usar demasiada memoria para las estructuras necesarias para soportar la virtualización). Al implementar una virtualización eficiente en el tiempo, el sistema operativo tendrá que depender del soporte de hardware, incluidas características de hardware como TLB (sobre las cuales aprenderemos a su debido tiempo).

Finalmente, un tercer objetivo de las máquinas virtuales es la protección. El sistema operativo debe asegurarse de proteger los procesos entre sí, así como el propio sistema operativo contra los pro-

<sup>2</sup>Este uso de la transparencia a veces resulta confuso; Algunos estudiantes piensan que "ser transparente" significa dejar todo al descubierto, es decir, cómo debería ser el gobierno.

Aquí significa lo contrario: que la ilusión proporcionada por el sistema operativo no debería ser visible para las aplicaciones. Por lo tanto, en el uso común, un sistema transparente es uno que es difícil de notar, no uno que responde a las solicitudes como lo estipula la Ley de Libertad de Información.

**CONSEJO: EL PRINCIPIO DE AISLAMIENTO**

El aislamiento es un principio clave en la construcción de sistemas confiables. Si dos entidades son adecuadamente aisladas unos de otros, esto implica que uno puede fallar sin afectar al otro. Los sistemas operativos se esfuerzan por aislar los procesos de entre sí y de esta manera evitar que uno dañe al otro. Al usar aislamiento de la memoria, el sistema operativo garantiza además que los programas en ejecución no puedan afectar el funcionamiento del sistema operativo subyacente. Algunos sistemas operativos modernos llevan el aislamiento aún más lejos, separando partes del sistema operativo de otras partes del sistema. el sistema operativo. Por tanto, dichos micronúcleos [BH70, R+89, S+03] pueden proporcionar una mayor confiabilidad que los diseños típicos de núcleo monolítico.

ceses. Cuando un proceso realiza una carga, un almacenamiento o una búsqueda de instrucciones, no debería poder acceder ni afectar de ninguna manera el contenido de la memoria de cualquier otro proceso o del propio sistema operativo (es decir, cualquier cosa fuera de su dirección espacio). La protección nos permite así entregar la propiedad del aislamiento. entre procesos; cada proceso debe ejecutarse en su propio capullo aislado, a salvo de los estragos de otros procesos defectuosos o incluso maliciosos.

En los próximos capítulos, centraremos nuestra exploración en los mecanismos básicos necesarios para virtualizar la memoria, incluido el hardware y el sistema operativo. soporte de sistemas. También investigaremos algunas de las políticas más relevantes que encontrará en los sistemas operativos, incluido cómo administrar espacio libre y qué páginas eliminar de la memoria cuando te quedas sin ella espacio. Al hacerlo, ampliaremos su comprensión de cómo un moderno sistema de memoria virtual realmente funciona<sup>3</sup>.

## 13.5 Resumen

Hemos visto la introducción de un importante subsistema del sistema operativo: la memoria virtual. El sistema VM es responsable de proporcionar la ilusión de un gran espacio de direcciones privado y escaso para los programas, que contienen todas sus instrucciones y datos. El sistema operativo, con alguna ayuda importante de hardware, tome cada una de estas referencias de memoria virtual y conviértalas en direcciones físicas, que pueden presentarse a la memoria física para obtener la información deseada. El sistema operativo hará esto para muchos procesos en una vez, asegurándose de proteger los programas entre sí, así como de proteger el sistema operativo. Todo el enfoque requiere una gran cantidad de mecanismos (muchos de maquinaria de bajo nivel), así como algunas políticas críticas para trabajar; Bueno Comience desde abajo hacia arriba, describiendo primero los mecanismos críticos. Y ¡Así procedemos!

---

<sup>3</sup>O te convenceremos de que abandones el curso. Pero espera; si lo logras a través de VM, podrás ¡Probablemente llegue hasta el final!

## APARTE: CADA DIRECCIÓN QUE VES ES VIRTUAL ¿Alguna vez

has escrito un programa en C que imprima un puntero? El valor que ve (un número grande, a menudo impreso en hexadecimal) es una dirección virtual.

¿Alguna vez te has preguntado dónde se encuentra el código de tu programa? También puedes imprimirla, y sí, si puedes imprimirla, también es una dirección virtual. De hecho, cualquier dirección que pueda ver como programador de un programa a nivel de usuario es una dirección virtual. Es sólo el sistema operativo, a través de sus complicadas técnicas de virtualización de la memoria, el que sabe en qué parte de la memoria física de la máquina se encuentran estas instrucciones y valores de datos. Así que nunca lo olvides: si imprimes una dirección en un programa, es virtual, una ilusión de cómo se disponen las cosas en la memoria; sólo el sistema operativo (y el hardware) conocen la verdad real.

Aquí hay un pequeño programa (va.c) que imprime las ubicaciones de la rutina main() (donde reside el código), el valor de un valor asignado al montón devuelto por malloc() y la ubicación de un número entero en la pila:

```

1 #include <stdio.h> 2 #include
<stdlib.h> 3 int main(int argc, char
*argv[]) {
4     printf("ubicación del código: %p\n", principal); printf("ubicación del
5         montón: %p\n", malloc(100e6)); entero x = 3; printf("ubicación de la pila: %p\n", &x);
6     devolver x;
7
8
9 }
```

Cuando se ejecuta en una Mac de 64 bits, obtenemos el siguiente resultado:

```

ubicación del código: 0x1095afe50
ubicación del montón: 0x1096008c0 ubicación del
montón: 0x7fff691aea64
```

A partir de esto, puede ver que el código aparece primero en el espacio de direcciones, luego en el montón, y la pila está completamente en el otro extremo de este gran espacio virtual. Todas estas direcciones son virtuales y serán traducidas por el sistema operativo y el hardware para recuperar valores de sus verdaderas ubicaciones físicas.

## Referencias

[BH70] "El núcleo de un sistema de multiprogramación" de Per Brinch Hansen. Communications of the ACM, 13:4, abril de 1970. El primer artículo que sugiere que el sistema operativo, o kernel, debería ser un sustrato mínimo y flexible para construir sistemas operativos personalizados; Este tema se retoma a lo largo de la historia de la investigación del sistema operativo.

[CV65] "Introducción y descripción general del sistema Multics" por FJ Corbato, VA Vyssotsky. Fall Joint Computer Conference, 1965. Un excelente artículo inicial de Multics. Aquí está la gran cita sobre el tiempo compartido: "El impetu por el tiempo compartido surgió por primera vez de los programadores profesionales debido a su constante frustración al depurar programas en instalaciones de procesamiento por lotes. Así, el objetivo original era compartir ordenadores para permitir el acceso simultáneo de varias personas y dar a cada una de ellas la ilusión de tener toda la máquina a su disposición".

[DV66] "Programación de semántica para computaciones multiprogramadas" por Jack B. Dennis, Earl C. Van Horn. Communications of the ACM, Volumen 9, Número 3, marzo de 1966. Uno de los primeros artículos (pero no el primero) sobre multiprogramación.

[L60] "Simbiosis Hombre-Computadora" por JCR Licklider. IRE Transactions on Human Factors in Electronics, HFE-1:1, marzo de 1960. Un artículo original sobre cómo las computadoras y las personas van a entrar en una era simbiótica; Claramente muy adelantado a su tiempo, pero de todos modos es una lectura fascinante.

[M62] "Sistemas informáticos de tiempo compartido" de J. McCarthy. Management and the Computer of the Future, MIT Press, Cambridge, MA, 1962. Probablemente el primer artículo registrado de McCarthy sobre el tiempo compartido. En otro artículo [M83], afirma haber estado pensando en la idea desde 1957.

McCarthy dejó el área de sistemas y pasó a convertirse en un gigante de la Inteligencia Artificial en Stanford, incluida la creación del lenguaje de programación LISP. Consulte la página de inicio de McCarthy para obtener más información: <http://www-formal.stanford.edu/jmc/>

[M+63] "Un sistema de depuración de tiempo compartido para una computadora pequeña" por J. McCarthy, S. Boilen, E. Fredkin, JCR Licklider. AFIPS '63 (primavera), Nueva York, NY, mayo de 1963. Un excelente ejemplo temprano de un sistema que intercambiaba la memoria del programa al "tambor" cuando el programa no se estaba ejecutando, y luego regresaba a la memoria "central" cuando estaba a punto de ejecutarse.

[M83] "Reminiscencias sobre la historia del tiempo compartido" de John McCarthy. 1983. Disponible: <http://www-formal.stanford.edu/jmc/history/timesharing/timesharing.html>. Una excelente nota histórica sobre de dónde podría haber surgido la idea del tiempo compartido, que incluye algunas dudas hacia quienes citan el trabajo de Strachey [S59] como el trabajo pionero en esta área.

[NS07] "Valgrind: un marco para instrumentación binaria dinámica de peso pesado" por N. Nethercote, J. Seward. PLDI 2007, San Diego, California, junio de 2007. Valgrind es un programa que salva vidas para aquellos que usan lenguajes inseguros como C. Lea este documento para conocer sus geniales técnicas de instrumentación binaria; es realmente bastante impresionante.

[R+89] "Mach: Un núcleo de software de sistema" por R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, M. Jones. COMPCON '89, febrero de 1989. Aunque no fue el primer proyecto sobre micronúcleos por se, el proyecto Mach en CMU fue muy conocido e influyente; todavía vive hoy en lo profundo de las entrañas de Mac OS X.

[S59] "Tiempo compartido en computadoras grandes y rápidas" por C. Strachey. Actas de la Conferencia Internacional sobre Procesamiento de Información, UNESCO, junio de 1959. Una de las primeras referencias sobre el tiempo compartido.

[S+03] "Mejora de la confiabilidad de los sistemas operativos de productos básicos" por MM Swift, BN Bershad, HM Levy. SOSP '03. El primer artículo que muestra cómo el pensamiento similar a un micronúcleo puede mejorar la confiabilidad del sistema operativo.

## Tarea (Código)

En esta tarea, aprenderemos sobre algunas herramientas útiles para examinar el uso de la memoria virtual en sistemas basados en Linux. Esto será sólo una breve pista de lo que es posible; Tendrás que profundizar más por tu cuenta para convertirte verdaderamente en un experto (¡como siempre!).

### Preguntas 1.

La primera herramienta de Linux que debe consultar es la sencilla herramienta gratuita.

Primero, escriba `man free` y lea toda la página del manual; es corto, ¡no te preocunes!

2. Ahora, ejecútelo libremente, quizás usando algunos de los argumentos que puedan ser útiles (por ejemplo, `-m`, para mostrar los totales de memoria en megabytes). ¿Cuánta memoria hay en su sistema? ¿Cuánto es gratis? ¿Estos números coinciden con tu intuición?
3. A continuación, cree un pequeño programa que utilice una cierta cantidad de memoria, llamado `memoria-user.c`. Este programa debe tomar un argumento de línea de comando: la cantidad de megabytes de memoria que utilizará.  
Cuando se ejecuta, debe asignar una matriz y transmitirla constantemente a través de la matriz, tocando cada entrada. El programa debería hacer esto de forma indefinida o, quizás, durante un cierto período de tiempo también especificado en la línea de comando.
4. Ahora, mientras ejecuta su programa de usuario de memoria , ejecute también (en una ventana de terminal diferente, pero en la misma máquina) la herramienta gratuita .  
¿Cómo cambian los totales de uso de memoria cuando se ejecuta el programa? ¿Qué tal cuando matas el programa de usuario de memoria ?  
¿Los números coinciden con sus expectativas? Pruebe esto para diferentes cantidades de uso de memoria. ¿Qué sucede cuando usas cantidades realmente grandes de memoria?
5. Probemos con una herramienta más, conocida como `pmap`. Dedique algo de tiempo y lea la página del manual de `pmap` en detalle.
6. Para usar `pmap`, debe conocer el ID del proceso que le interesa. Por lo tanto, primero ejecute `ps auxw` para ver una lista de todos los procesos; luego, elige uno interesante, como un navegador. También puede usar su programa de usuario de memoria en este caso (de hecho, incluso puede hacer que ese programa llame a `getpid()` e imprima su PID para su conveniencia).
7. Ahora ejecute `pmap` en algunos de estos procesos, usando varios indicadores (como `-X`) para revelar muchos detalles sobre el proceso. ¿Qué ves?  
¿Cuántas entidades diferentes componen un espacio de direcciones moderno, a diferencia de nuestra simple concepción de código/pila/montón?
8. Finalmente, ejecutemos `pmap` en su programa de usuario de memoria , con diferentes cantidades de memoria utilizada. ¿Qué ves aquí? ¿El resultado de `pmap` coincide con sus expectativas?

## Interludio: API de memoria

En este interludio, analizamos las interfaces de asignación de memoria en sistemas UNIX . Las interfaces proporcionadas son bastante simples y, por lo tanto, el capítulo es breve y directo<sup>1</sup> . El principal problema que abordamos es este:

### CRUX: CÓMO ASIGNAR Y GESTIONAR LA MEMORIA

En los programas UNIX/C , comprender cómo asignar y administrar la memoria es fundamental para crear un software sólido y confiable. ¿Qué interfaces se utilizan comúnmente? ¿Qué errores se deben evitar?

#### 14.1 Tipos de memoria Al ejecutar

un programa en C, se asignan dos tipos de memoria. La primera se llama memoria de pila , y el compilador gestiona implícitamente sus asignaciones y desasignaciones para usted, el programador; por esta razón a veces se la llama memoria automática .

Declarar memoria en la pila en C es fácil. Por ejemplo, digamos que necesita algo de espacio en una función func() para un número entero, llamado x. Para declarar tal fragmento de memoria, simplemente haga algo como esto:

```
función vacía() { int
    x; // declara un número entero en la pila
    ...
}
```

El compilador hace el resto, asegurándose de dejar espacio en la pila cuando llama a func(). Cuando regresa de la función, el compilador desasigna la memoria por usted; por lo tanto, si desea que alguna información permanezca más allá de la invocación de la llamada, será mejor que no deje esa información en la pila.

---

<sup>1</sup>De hecho, ¡esperamos que todos los capítulos lo sean! Pero creemos que este es más corto y puntiagudo.

Es esta necesidad de memoria de larga duración la que nos lleva al segundo tipo de memoria, llamada memoria de montón , donde todas las asignaciones y desasignaciones son manejadas explícitamente por usted, el programador. ¡Una gran responsabilidad, sin duda! Y ciertamente la causa de muchos errores. Pero si tienes cuidado y prestas atención, utilizarás dichas interfaces correctamente y sin demasiados problemas. A continuación se muestra un ejemplo de cómo se podría asignar un número entero en el montón:

```
función vacía() {
    int *x = (int *) malloc(tamañode(int));
    ...
}
```

Un par de notas sobre este pequeño fragmento de código. Primero, puede notar que tanto la asignación de pila como de montón ocurren en esta línea: primero, el compilador sabe que debe dejar espacio para un puntero a un número entero cuando ve su declaración de dicho puntero (`int *x`); posteriormente, cuando el programa llama a `malloc()`, solicita espacio para un número entero en el montón; la rutina devuelve la dirección de dicho número entero (en caso de éxito, o `NULL` en caso de error), que luego se almacena en la pila para que lo utilice el programa.

Debido a su naturaleza explícita y a su uso más variado, la memoria dinámica presenta más desafíos tanto para los usuarios como para los sistemas. Por lo tanto, es el foco del resto de nuestra discusión.

## 14.2 La llamada a `malloc()`

La llamada a `malloc()` es bastante simple: le pasas un tamaño solicitando algo de espacio en el montón, y tiene éxito y te devuelve un puntero al espacio recién asignado, o falla y devuelve `NULL`<sup>2</sup>.

La página del manual muestra lo que debe hacer para usar `malloc`; tipo hombre `malloc` en la línea de comando y verá:

```
#incluir <stdlib.h>
...
void *malloc(tamaño_t tamaño);
```

A partir de esta información, puede ver que todo lo que necesita hacer es incluir el archivo de encabezado `stdlib.h` para usar `malloc`. De hecho, ni siquiera necesitas hacer esto, ya que la biblioteca C, con la que todos los programas C se vinculan de forma predeterminada, tiene el código para `malloc()` dentro; agregar el encabezado solo le permite al compilador verificar si está llamando a `malloc()` correctamente (por ejemplo, pasándole la cantidad correcta de argumentos, del tipo correcto).

El único parámetro que toma `malloc()` es de tamaño de tipo `t`, que simplemente describe cuántos bytes necesita. Sin embargo, la mayoría de los programadores no escriben aquí un número directamente (como 10); de hecho, se consideraría de mala educación hacerlo. En su lugar, se incluyen varias rutinas y macros.

---

<sup>2</sup>Tenga en cuenta que `NULL` en C no es nada especial, sólo una macro para el valor cero.

**CONSEJO: CUANDO DUDA , PRUEBE Si no**

está seguro de cómo se comporta alguna rutina u operador que está utilizando, no hay nada mejor que probarlo y asegurarse de que se comporta como espera . Si bien leer las páginas del manual u otra documentación es útil, lo que importa es cómo funciona en la práctica. ¡Escribe un código y pruébalo! Sin duda, esa es la mejor manera de asegurarse de que su código se comporte como desea. De hecho, ¡eso es lo que hicimos para verificar que las cosas que decíamos sobre sizeof() fueran realmente ciertas!

utilizado. Por ejemplo, para asignar espacio para un valor de punto flotante de doble precisión, simplemente haga esto:

```
doble *d = (doble *) malloc(tamañode(doble));
```

¡Vaya, eso es mucho duplicar! Esta invocación de malloc() utiliza el operador sizeof() para solicitar la cantidad correcta de espacio; En C, esto generalmente se considera un operador en tiempo de compilación, lo que significa que el tamaño real se conoce en el momento de la compilación y, por lo tanto, se sustituye un número (en este caso, 8, por un doble) como argumento de malloc(). Por esta razón, sizeof() se considera correctamente como un operador y no como una llamada a función (una llamada a función se llevaría a cabo en tiempo de ejecución).

También puedes pasar el nombre de una variable (y no sólo un tipo) a sizeof(), pero en algunos casos es posible que no obtengas los resultados deseados, así que ten cuidado. Por ejemplo, veamos el siguiente fragmento de código:

```
int *x = malloc(10 * tamaño de(int)); printf("%d\n",
tamañode(x));
```

En la primera línea, hemos declarado espacio para una matriz de 10 números enteros, lo cual está muy bien. Sin embargo, cuando usamos sizeof() en la siguiente línea, devuelve un valor pequeño, como 4 (en máquinas de 32 bits) u 8 (en máquinas de 64 bits). La razón es que en este caso, sizeof() piensa que simplemente estamos preguntando qué tan grande es un puntero a un número entero, no cuánta memoria hemos asignado dinámicamente. Sin embargo, a veces sizeof() funciona como es de esperar:

```
intx[10];
printf("%d\n", tamañode(x));
```

En este caso, hay suficiente información estática para que el compilador sepa que se han asignado 40 bytes.

Otro lugar donde hay que tener cuidado es con las cuerdas. Al declarar espacio para una cadena, use el siguiente modismo: malloc(strlen(s) + 1), que obtiene la longitud de la cadena usando la función strlen() y le agrega 1 para dejar espacio para el final. carácter de cadena. Usar sizeof() puede generar problemas aquí.

También puedes notar que malloc() devuelve un puntero para escribir void. Hacerlo es simplemente la forma en C de devolver una dirección y dejar que el programador decida qué hacer con ella. El programador ayuda además utilizando lo que se llama un reparto; En nuestro ejemplo anterior, el programador convierte el tipo de retorno de malloc() en un puntero a un doble. La transmisión realmente no logra nada, aparte de decirle al compilador y a otros programadores que podrían estar leyendo su código: "sí, sé lo que estoy haciendo". Al emitir el resultado de malloc(), el programador simplemente está dando cierta tranquilidad; el yeso no es necesario para la corrección.

### 14.3 La llamada gratuita()

Resulta que asignar memoria es la parte fácil de la ecuación; saber cuándo, cómo e incluso si liberar memoria es la parte difícil. Para liberar memoria dinámica que ya no está en uso, los programadores simplemente llaman a free(): int \*x = malloc(10 \* sizeof(int));

...

libre(x); La

rutina toma un argumento, un puntero devuelto por malloc().

Por lo tanto, como podrá observar, el tamaño de la región asignada no lo pasa el usuario y debe ser rastreado por la propia biblioteca de asignación de memoria.

### 14.4 Errores comunes

Hay una serie de errores comunes que surgen en el uso de malloc() y free(). Éstos son algunos de los que hemos visto una y otra vez al impartir el curso universitario de sistemas operativos. Todos estos ejemplos se compilan y ejecutan sin apenas pío del compilador; Si bien compilar un programa en C es necesario para construir un programa en C correcto, está lejos de ser suficiente, como aprenderá (a menudo de la manera más difícil).

De hecho, la gestión correcta de la memoria ha sido un problema tal que muchos lenguajes más nuevos admiten la gestión automática de la memoria. En dichos lenguajes, mientras llamas a algo similar a malloc() para asignar memoria (generalmente nueva o algo similar para asignar un nuevo objeto), nunca tienes que llamar a algo para liberar espacio; más bien, se ejecuta un recolector de basura y descubre a qué memoria ya no tiene referencias y la libera.

Olvidar asignar memoria Muchas rutinas

esperan que se asigne memoria antes de llamarlas. Por ejemplo, la rutina strcpy(dst, src) copia una cadena desde un puntero de origen a un puntero de destino. Sin embargo, si no tienes cuidado, puedes hacer esto: char \*src = "hello"; carbón \*dst; // ¡ups! strcpy no asignado (dst,

src); // segmentar y morir

**CONSEJO: SE COMPILO O SE EJECUTÓ = ES CORRECTO Sólo**

porque un programa se compiló (!) o incluso se ejecutó una o muchas veces correctamente no significa que el programa sea correcto . Es posible que muchos eventos hayan conspirado para llevarte a un punto en el que crees que funciona, pero luego algo cambia y se detiene. Una reacción común de los estudiantes es decir (o gritar):

"¡Pero funcionó antes!" y luego culpar al compilador, al sistema operativo, al hardware o incluso (nos atrevemos a decirlo) al profesor. Pero el problema suele estar justo donde cree que estaría, en su código. Póngase a trabajar y depúrelo antes de culpar a esos otros componentes.

Cuando ejecuta este código, probablemente provocará un error de segmentación<sup>3</sup> , que es , un término elegante para HICISTE ALGO MAL CON LA MEMORIA, PROGRAMADOR TONTO Y ESTOY ENOJADO.

En este caso, el código adecuado podría verse así:

```
char *src = "hola"; char *dst = (char
*) malloc(strlen(src) + 1); strcpy(dst, origen); // funciona correctamente
```

Alternativamente, puedes usar strdup() y hacerle la vida aún más fácil.  
Lea la página de manual de strdup para obtener más información.

No asignar suficiente memoria Un error

relacionado es no asignar suficiente memoria, lo que a veces se denomina desbordamiento del búfer. En el ejemplo anterior, un error común es dejar casi suficiente espacio para el búfer de destino.

```
char *src = "hola"; char *dst = (char
*) malloc(strlen(src)); // ¡demasiado pequeño! strcpy(dst, origen); // funciona correctamente
```

Por extraño que parezca, dependiendo de cómo se implemente malloc y muchos otros detalles, este programa a menudo se ejecutará aparentemente correctamente. En algunos casos, cuando se ejecuta la copia de la cadena, escribe un byte más allá del final del espacio asignado, pero en algunos casos esto es inofensivo, tal vez sobrescribiendo una variable que ya no se usa. En algunos casos, estos desbordamientos pueden ser increíblemente dañinos y, de hecho, son la fuente de muchas vulnerabilidades de seguridad en los sistemas [W06]. En otros casos, la biblioteca malloc asignó un poco de espacio adicional de todos modos y, por lo tanto, su programa en realidad no garabatea el valor de alguna otra variable y funciona bastante bien. Incluso en otros casos, el programa fallará y fallará. Y así aprendemos otra lección valiosa: aunque se haya ejecutado correctamente una vez, no significa que sea correcto.

---

<sup>3</sup>Aunque suene arcano, pronto aprenderá por qué se permite un acceso a la memoria tan ilegal. llamada falla de segmentación; Si eso no es un incentivo para seguir leyendo, ¿cuál lo es?

Olvidar inicializar la memoria asignada Con este error, llama a malloc() correctamente, pero olvida ingresar algunos valores en el tipo de datos recién asignado. ¡No hagas esto! Si lo olvida, su programa eventualmente encontrará una lectura no inicializada, donde lee del montón algunos datos de valor desconocido. ¿Quién sabe qué podría haber ahí dentro? Si tiene suerte, algún valor tal que el programa aún funcione (por ejemplo, cero). Si no tienes suerte, algo aleatorio y dañino.

#### Olvidar liberar memoria Otro error

común se conoce como pérdida de memoria y ocurre cuando olvida liberar memoria. En aplicaciones o sistemas de larga ejecución (como el propio sistema operativo), esto es un gran problema, ya que la pérdida lenta de memoria eventualmente lleva a que uno se quede sin memoria, momento en el cual es necesario reiniciar. Por lo tanto, en general, cuando haya terminado con una parte de la memoria, debe asegurarse de liberarla. Tenga en cuenta que usar un lenguaje de recolección de basura no ayuda aquí: si todavía tiene una referencia a algún fragmento de memoria, ningún recolector de basura la liberará jamás y, por lo tanto, las pérdidas de memoria siguen siendo un problema incluso en lenguajes más modernos.

En algunos casos, puede parecer que no llamar a free() es razonable. Por ejemplo, su programa dura poco y pronto finalizará; en este caso, cuando el proceso finaliza, el sistema operativo limpiará todas las páginas asignadas y, por lo tanto, no se producirá ninguna pérdida de memoria. Si bien esto ciertamente “funciona” (ver el aparte en la página 7), probablemente sea un mal hábito de desarrollar, así que tenga cuidado al elegir dicha estrategia. A largo plazo, uno de sus objetivos como programador es desarrollar buenos hábitos; Uno de esos hábitos es comprender cómo gestiona la memoria y (en lenguajes como C), liberar los bloques que ha asignado. Incluso si puede evitarlo, probablemente sea bueno acostumbrarse a liberar todos y cada uno de los bytes que asigne explícitamente.

#### Liberar memoria antes de terminar A veces un programa libera

memoria antes de terminar de usarla; Tal error se llama puntero colgante y, como puedes adivinar, también es algo malo. El uso posterior puede bloquear el programa o sobrescribir la memoria válida (por ejemplo, llamaste a free(), pero luego llamaste a malloc() nuevamente para asignar algo más, que luego recicla la memoria liberada erróneamente).

#### Liberar memoria repetidamente Los

programas a veces también liberan memoria más de una vez; esto se conoce como el doble gratis. El resultado de hacerlo no está definido. Como puedes imaginar, la biblioteca de asignación de memoria puede confundirse y hacer todo tipo de cosas raras; Los accidentes son un resultado común.

**APARTE: POR QUÉ NO SE PIERDE MEMORIA UNA VEZ QUE SU PROCESO SALE**

Cuando escribes un programa de corta duración, puedes asignar algo de espacio usando malloc(). El programa se ejecuta y está a punto de completarse: ¿es necesario llamar a free() varias veces justo antes de salir? Si bien parece incorrecto no hacerlo, no se “perderá” ningún recuerdo en ningún sentido real. La razón es simple: en realidad existen dos niveles de gestión de memoria en el sistema.

El primer nivel de gestión de la memoria lo realiza el sistema operativo, que entrega memoria a los procesos cuando se ejecutan y la recupera cuando los procesos salen (o mueren). El segundo nivel de gestión está dentro de cada proceso, por ejemplo dentro del montón cuando llamas a malloc() y free(). Incluso si no llama a free() (y, por lo tanto, pierde memoria en el montón), el sistema operativo recuperará toda la memoria del proceso (incluidas las páginas de código, pila y, como corresponde aquí, montón) cuando el programa ha terminado de ejecutarse. No importa cuál sea el estado de su montón en su espacio de direcciones, el sistema operativo recupera todas esas páginas cuando el proceso finaliza, asegurando así que no se pierda memoria a pesar de que no la haya liberado.

Por lo tanto, para programas de corta duración, la pérdida de memoria a menudo no causa ningún problema operativo (aunque puede considerarse de mala forma). Cuando escribe un servidor de larga duración (como un servidor web o un sistema de administración de bases de datos, que nunca sale), la pérdida de memoria es un problema mucho mayor y, eventualmente, provocará una falla cuando la aplicación se quede sin memoria. . Y, por supuesto, la pérdida de memoria es un problema aún mayor dentro de un programa en particular: el propio sistema operativo. Mostrándonos una vez más: aquellos que escriben el código del kernel tienen el trabajo más duro de todos...

**Llamar a free() incorrectamente Un**

último problema que analizamos es la llamada incorrecta a free(). Después de todo, free() espera que solo le pases uno de los punteros que recibiste anteriormente de malloc(). Cuando se pasa algún otro valor, pueden suceder (y suceden) cosas malas. Por lo tanto, estas liberaciones inválidas son peligrosas y, por supuesto, también deben evitarse.

**Resumen**

Como puede ver, hay muchas formas de abusar de la memoria. Debido a los frecuentes errores de memoria, se ha desarrollado toda una ecosfera de herramientas para ayudar a encontrar dichos problemas en el código. Echa un vistazo a purificar [HJ92] y valgrind [SN05]; Ambos son excelentes para ayudarte a localizar el origen de sus problemas relacionados con la memoria. Una vez que se acostumbre a utilizar estas poderosas herramientas, se preguntará cómo sobrevivió sin ellas.

#### 14.5 Compatibilidad con el sistema operativo subyacente

Es posible que hayas notado que no hemos estado hablando de llamadas al sistema cuando hablamos de malloc() y free(). La razón de esto es simple: no son llamadas al sistema, sino llamadas a la biblioteca. Por lo tanto, la biblioteca malloc administra el espacio dentro de su espacio de direcciones virtuales, pero está construida sobre algunas llamadas al sistema que llaman al sistema operativo para solicitar más memoria o liberar algo al sistema.

Una de esas llamadas al sistema se llama brk, que se utiliza para cambiar la ubicación de la interrupción del programa: la ubicación del final del montón. Toma un argumento (la dirección de la nueva ruptura) y, por lo tanto, aumenta o disminuye el tamaño del montón en función de si la nueva ruptura es mayor o menor que la ruptura actual. A una llamada adicional sbrk se le pasa un incremento, pero por lo demás tiene un propósito similar.

Tenga en cuenta que nunca debe llamar directamente a brk o sbrk. Son utilizados por la biblioteca de asignación de memoria; si intentas utilizarlos, es probable que algo salga (horriblemente) mal. Limítense a malloc() y free() en su lugar.

Finalmente, también puedes obtener memoria del sistema operativo mediante la llamada mmap(). Al pasar los argumentos correctos, mmap() puede crear una región de memoria anónima dentro de su programa, una región que no está asociada con ningún archivo en particular sino con un espacio de intercambio, algo que discutiremos en detalle más adelante en la memoria virtual. Esta memoria también puede tratarse como un montón y gestionarse como tal. Lea la página del manual de mmap() para más detalles.

#### 14.6 Otras llamadas

Hay algunas otras llamadas que admite la biblioteca de asignación de memoria. Por ejemplo, calloc() asigna memoria y también la pone a cero antes de regresar; esto evita algunos errores en los que se supone que la memoria está puesta a cero y se olvida inicializarla usted mismo (consulte el párrafo sobre "lecturas no inicializadas" más arriba). La rutina realloc() también puede ser útil cuando ha asignado espacio para algo (por ejemplo, una matriz) y luego necesita agregarle algo: realloc() crea una nueva región más grande de memoria, copia la región anterior en y devuelve el puntero a la nueva región.

#### 14.7 Resumen

Hemos introducido algunas de las API que se ocupan de la asignación de memoria. Como siempre, acabamos de cubrir lo básico; más detalles están disponibles en otros lugares. Lea el libro C [KR88] y Stevens [SR05] (Capítulo 7) para obtener más información. Para obtener un artículo moderno y interesante sobre cómo detectar y corregir muchos de estos problemas automáticamente, consulte Novark et al. [N+07]; Este documento también contiene un buen resumen de problemas comunes y algunas ideas interesantes sobre cómo encontrarlos y solucionarlos.

## Referencias

[HJ92] "Purificar: Detección rápida de pérdidas de memoria y errores de acceso" por R. Hastings, B. Joyce. USENIX Invierno '92. El artículo detrás de la genial herramienta Purify, ahora un producto comercial.

[KR88] "El lenguaje de programación C" de Brian Kernighan, Dennis Ritchie. Prentice-Hall 1988. El libro de C, escrito por los desarrolladores de C. Léalo una vez, programe un poco, luego léalo de nuevo y luego manténgalo cerca de su escritorio o dondequiero que programe.

[N+07] "Exterminador: Corrección automática de errores de memoria con alta probabilidad" por G. Novark, ED Berger, BG Zorn. PLDI 2007, San Diego, California. Un artículo interesante sobre cómo encontrar y corregir errores de memoria automáticamente y una excelente descripción general de muchos errores comunes en programas C y C++. Una versión ampliada de este documento está disponible en CACM (Volumen 51, Número 12, diciembre de 2008).

[SN05] "Uso de Valgrind para detectar errores de valores indefinidos con precisión de bits" por J. Seward, N. Nethercote. USENIX '05. Cómo utilizar valgrind para encontrar ciertos tipos de errores.

[SR05] "Programación avanzada en el entorno UNIX " por W. Richard Stevens, Stephen A. Rago. Addison-Wesley, 2005. Lo hemos dicho antes, lo diremos de nuevo: lea este libro muchas veces y úselo como referencia siempre que tenga dudas. Los autores siempre se sorprenden de cómo cada vez que leen algo de este libro, aprenden algo nuevo, incluso después de muchos años de programación en C.

[W06] "Encuesta sobre ataques y contramedidas de desbordamiento de búfer" por T. Werthman. Disponible: [www.nds.rub.de/lehre/seminar/SS06/Werthmann\\_BufferOverflow.pdf](http://www.nds.rub.de/lehre/seminar/SS06/Werthmann_BufferOverflow.pdf). Un buen estudio sobre los desbordamientos del buffer y algunos de los problemas de seguridad que causan. Se refiere a muchas de las hazañas famosas.

## Tarea (Código)

En esta tarea, se familiarizará un poco con la asignación de memoria. Primero, escribirás algunos programas con errores (¡divertido!). Luego, utilizará algunas herramientas que le ayudarán a encontrar los errores que insertó. Luego, te darás cuenta de lo maravillosas que son estas herramientas y las utilizarás en el futuro, haciéndote así más feliz y productivo. Las herramientas son el depurador (por ejemplo, gdb) y un detector de errores de memoria llamado valgrind [SN05].

## Preguntas

1. Primero, escriba un programa simple llamado null.c que cree un puntero a un número entero, lo establezca en NULL y luego intente desreferenciarlo. Compile esto en un ejecutable llamado nulo. ¿Qué sucede cuando ejecutas este programa?
2. A continuación, compile este programa con información de símbolos incluida (con el indicador -g). Al hacerlo, coloquemos más información en el ejecutable, permitiendo al depurador acceder a información más útil sobre nombres de variables y similares. Ejecute el programa bajo el depurador escribiendo gdb null y luego, una vez que gdb se esté ejecutando, escribiendo ejecutar. ¿Qué te muestra gdb?
3. Finalmente, use la herramienta valgrind en este programa. Usaremos la herramienta memcheck que forma parte de valgrind para analizar lo que sucede. Ejecute esto escribiendo lo siguiente: valgrind --leak-check=yes null. ¿Qué sucede cuando ejecutas esto? ¿Puedes interpretar el resultado de la herramienta?
4. Escriba un programa simple que asigne memoria usando malloc() pero olvide liberarla antes de salir. ¿Qué sucede cuando se ejecuta este programa? ¿Puedes usar gdb para encontrar algún problema? ¿Qué tal valgrind (de nuevo con la bandera --leak-check=yes)?
5. Escriba un programa que cree una matriz de números enteros llamados datos de tamaño 100 usando malloc; luego, establezca datos [100] en cero. ¿Qué sucede cuando ejecutas este programa? ¿Qué sucede cuando ejecutas este programa usando valgrind? ¿Es correcto el programa?
6. Cree un programa que asigne una matriz de números enteros (como arriba), los libere y luego intente imprimir el valor de uno de los elementos de la matriz. ¿Se ejecuta el programa? ¿Qué sucede cuando usas valgrind en él?
7. Ahora pase un valor curioso a free (por ejemplo, un puntero en el medio de la matriz que asignó anteriormente). ¿Lo que sucede? ¿Necesitas herramientas para encontrar este tipo de problema?

8. Pruebe algunas de las otras interfaces para la asignación de memoria. Por ejemplo, cree una estructura de datos simple similar a un vector y rutinas relacionadas que utilicen realloc() para administrar el vector. Utilice una matriz para almacenar los elementos de los vectores; cuando un usuario agrega una entrada al vector, use realloc() para asignarle más espacio. ¿Qué tan bien lo hace? tal vector realizar? ¿Cómo se compara con una lista vinculada? Usar valgrind para ayudarle a encontrar errores.
9. Dedique más tiempo a leer sobre el uso de gdb y valgrind. Conocer sus herramientas es fundamental; Pasa el tiempo y aprende cómo llegar a ser. un depurador experto en el entorno UNIX y C.

## Mecanismo: traducción de direcciones

Al desarrollar la virtualización de la CPU, nos centramos en un mecanismo general conocido como ejecución directa limitada (o LDE). La idea detrás de LDE es simple: en su mayor parte, dejar que el programa se ejecute directamente en el hardware; sin embargo, en ciertos momentos clave (como cuando un proceso emite una llamada al sistema o se produce una interrupción del temporizador), haga arreglos para que el sistema operativo participe y se asegure de que suceda lo "correcto". Por lo tanto, el sistema operativo, con un poco de soporte de hardware, hace todo lo posible por apartarse del programa en ejecución para ofrecer una virtualización eficiente; sin embargo, al interponerse en esos puntos críticos en el tiempo, el sistema operativo garantiza que mantiene el control sobre el hardware. La eficiencia y el control juntos son dos de los principales objetivos de cualquier sistema operativo moderno.

Al virtualizar la memoria, seguiremos una estrategia similar, logrando eficiencia y control al tiempo que proporcionamos la virtualización deseada. La eficiencia dicta que hagamos uso del soporte de hardware, que al principio será bastante rudimentario (p. ej., sólo unos pocos registros) pero crecerá hasta volverse bastante complejo (p. ej., TLB, soporte de tablas de páginas, etc., a medida que vaya avanzando). ya veremos). El control implica que el sistema operativo garantiza que ninguna aplicación pueda acceder a ninguna memoria que no sea la suya propia; por lo tanto, para proteger las aplicaciones entre sí y el sistema operativo de las aplicaciones, aquí también necesitaremos ayuda del hardware. Finalmente, necesitaremos un poco más del sistema VM, en términos de flexibilidad; Específicamente, nos gustaría que los programas pudieran usar sus espacios de direcciones de la forma que quisieran, haciendo así que el sistema sea más fácil de programar. Y así llegamos al quid refinado:

### EL CRUX: CÓMO

VIRTUALIZAR LA MEMORIA DE FORMA EFICIENTE Y FLEXIBLE ¿Cómo podemos construir una virtualización eficiente de la memoria? ¿Cómo proporcionamos la flexibilidad que necesitan las aplicaciones? ¿Cómo mantenemos el control sobre a qué ubicaciones de memoria puede acceder una aplicación y, por lo tanto, garantizamos que los accesos a la memoria de la aplicación estén restringidos adecuadamente? ¿Cómo hacemos todo esto de manera eficiente?

La técnica genérica que utilizaremos, que puedes considerar un añadido a nuestro enfoque general de ejecución directa limitada, es algo que es Esto se conoce como traducción de direcciones basada en hardware, o simplemente traducción de direcciones para abreviar. Con la traducción de direcciones, el hardware transforma cada acceso a la memoria (por ejemplo, una instrucción para buscar, cargar o almacenar), cambiando la dirección virtual proporcionada por la instrucción a una dirección física donde se encuentra realmente la información deseada. Así, en todos y cada uno de los recuerdos referencia, el hardware realiza una traducción de direcciones para redirigir referencias de memoria de la aplicación a sus ubicaciones reales en la memoria.

Por supuesto, el hardware por sí solo no puede virtualizar la memoria, ya que sólo proporciona el mecanismo de bajo nivel para hacerlo de manera eficiente. El sistema operativo debe obtener involucrado en puntos clave para configurar el hardware de modo que se realicen las traducciones correctas; por lo tanto debe gestionar la memoria, manteniendo un registro de qué ubicaciones son libres y están en uso, e intervenir juiciosamente para mantener el control sobre cómo se utiliza la memoria.

Una vez más el objetivo de todo este trabajo es crear una hermosa ilusión: que el programa tiene su propia memoria privada, donde se almacena su propio código. y los datos residen. Detrás de esa realidad virtual se esconde la fea verdad física: que muchos programas en realidad comparten memoria al mismo tiempo, como la CPU (o CPU) cambia entre ejecutar un programa y el siguiente. A través de la virtualización, el sistema operativo (con la ayuda del hardware) se vuelve feo convertir la realidad de la máquina en una abstracción útil, poderosa y fácil de usar.

## 15.1 Supuestos

Nuestros primeros intentos de virtualizar la memoria serán muy sencillos, casi ridículamente. Adelante, ríe todo lo que quieras; muy pronto será el sistema operativo riéndose de ti, cuando intentas comprender los entresijos de los TLB, tablas de páginas de varios niveles y otras maravillas técnicas. No me gusta la idea ¿El sistema operativo se ríe de ti? Bueno, entonces puedes que no tengas suerte; eso es solo cómo funciona el sistema operativo.

Especificamente, asumiremos por ahora que el espacio de direcciones del usuario debe colocarse de forma contigua en la memoria física. También asumiremos, por simplicidad, que el tamaño del espacio de direcciones no es demasiado grande; específicamente, que es menor que el tamaño de la memoria física. Finalmente, también asumiremos que cada espacio de direcciones tiene exactamente el mismo tamaño. No se preocupe si estas suposiciones parecen poco realistas; los iremos relajando a medida que avanzamos consiguiendo así una Virtualización realista de la memoria.

## 15.2 Un ejemplo

Para comprender mejor lo que debemos hacer para implementar la traducción de direcciones y por qué necesitamos dicho mecanismo, veamos un ejemplo simple. Imagine que hay un proceso cuyo espacio de direcciones es el indicado en Figura 15.1. Lo que vamos a examinar aquí es una secuencia de código corta. que carga un valor de la memoria, lo incrementa en tres y luego lo almacena el valor de nuevo en la memoria. Puedes imaginar que la representación en lenguaje C de este código podría verse así:

## CONSEJO: LA INTERPOSICIÓN ES PODEROSA La

interposición es una técnica genérica y poderosa que a menudo se utiliza con gran efecto en los sistemas informáticos. Al virtualizar la memoria, el hardware se interpondrá en cada acceso a la memoria y traducirá cada dirección virtual emitida por el proceso a una dirección física donde realmente se almacena la información deseada. Sin embargo, la técnica general de interposición es mucho más aplicable; de hecho, se puede interponer casi cualquier interfaz bien definida para agregar nuevas funciones o mejorar algún otro aspecto del sistema. Uno de los beneficios habituales de este enfoque es la transparencia; la interposición a menudo se realiza sin cambiar la interfaz del cliente, por lo que no requiere cambios en dicho cliente.

```
función vacía() {
    entero x = 3000; // gracias, Perry. x = x + 3; // línea de
    código que nos interesa
    ...
}
```

El compilador convierte esta línea de código en ensamblador, que podría verse así (en ensamblador x86). Utilice objdump en Linux u otool en Mac para desmontarlo:

128: movl 0x0(%ebx), %eax 132: addl \$0x03, %eax 135: movl %eax, 0x0(%ebx)	;cargar 0+ebx en eax ;agregar 3 al registro eax ;almacenar eax nuevamente en mem
--	--

Este fragmento de código es relativamente sencillo; supone que la dirección de x se ha colocado en el registro ebx y luego carga el valor en esa dirección en el registro de propósito general eax usando la instrucción movl (para movimiento de "palabra larga"). La siguiente instrucción suma 3 a eax y la instrucción final almacena el valor en eax en la memoria en esa misma ubicación.

En la Figura 15.1 (página 4), observe cómo tanto el código como los datos se disponen en el espacio de direcciones del proceso; la secuencia de código de tres instrucciones se encuentra en la dirección 128 (en la sección de código cerca de la parte superior) y el valor de la variable x en la dirección 15 KB (en la pila cerca de la parte inferior). En la figura, el valor inicial de x es 3000, como se muestra en su ubicación en la pila.

Cuando se ejecutan estas instrucciones, desde la perspectiva del proceso, se producen los siguientes accesos a la memoria.

- Obtener instrucción en la dirección 128 •
- Ejecutar esta instrucción (cargar desde la dirección 15 KB) •
- Obtener instrucción en la dirección 132 •
- Ejecutar esta instrucción (sin referencia de memoria) • Obtener la instrucción en la dirección 135 • Ejecutar esta instrucción (almacenar en la dirección 15 KB)

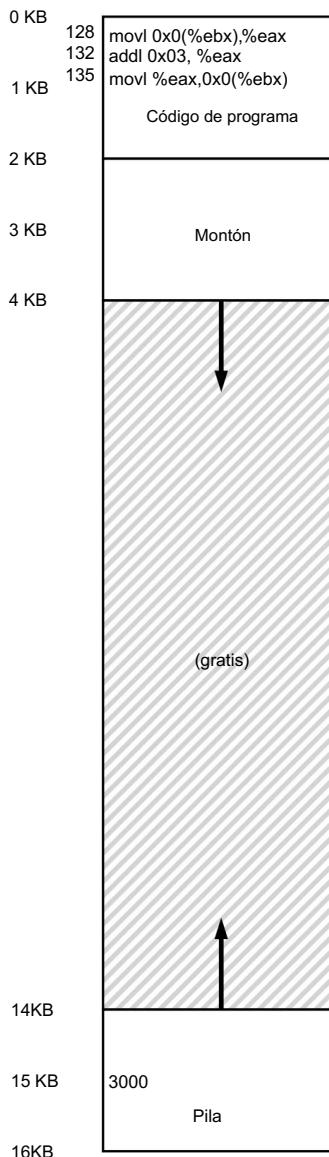


Figura 15.1: Un proceso y su espacio de direcciones

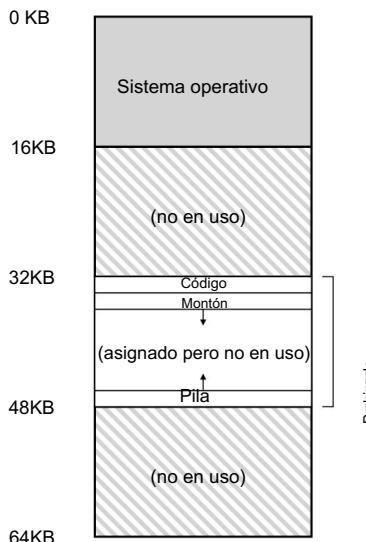


Figura 15.2: Memoria física con un único proceso reubicado Desde la

perspectiva del programa, su espacio de direcciones comienza en la dirección 0 y crece hasta un máximo de 16 KB; todas las referencias de memoria que genera deben estar dentro de estos límites. Sin embargo, para virtualizar la memoria, el sistema operativo quiere colocar el proceso en otro lugar de la memoria física, no necesariamente en la dirección 0. Por lo tanto, tenemos el problema: ¿cómo podemos reubicar este proceso en la memoria de una manera que sea transparente para el usuario? ¿proceso? ¿Cómo podemos proporcionar la ilusión de un espacio de direcciones virtuales que comienza en 0, cuando en realidad el espacio de direcciones está ubicado en alguna otra dirección física?

En la figura 15.2 se encuentra un ejemplo de cómo podría verse la memoria física una vez que el espacio de direcciones de este proceso se ha colocado en la memoria. En la figura, puede ver el sistema operativo usando la primera ranura de memoria física para sí mismo y que ha reubicado el proceso del ejemplo anterior en la ranura que comienza en la dirección de memoria física de 32 KB. Los otros dos slots están libres (16 KB-32 KB y 48 KB-64 KB).

### 15.3 Reubicación dinámica (basada en hardware) Para comprender

mejor la traducción de direcciones basada en hardware, primero analizaremos su primera encarnación. En las primeras máquinas de tiempo compartido de finales de la década de 1950 se introdujo una idea sencilla denominada base y límites; la técnica también se conoce como reubicación dinámica; Usaremos ambos términos indistintamente [SS74].

Especificamente, necesitaremos dos registros de hardware dentro de cada CPU: uno se llama registro base y el otro límites (a veces llamado registro límite). Este par de bases y límites nos va a permitir colocar el

**APARTE: REUBICACIÓN BASADA EN SOFTWARE** En los

primeros días, antes de que surgiera el soporte de hardware, algunos sistemas realizaban una forma burda de reubicación únicamente a través de métodos de software. La técnica básica se conoce como reubicación estática, en la que una pieza de software conocida como cargador toma un ejecutable que está a punto de ejecutarse y reescribe sus direcciones en el desplazamiento deseado en la memoria física.

Por ejemplo, si una instrucción se cargó desde la dirección 1000 en un registro (por ejemplo, `movl 1000, %eax`), y el espacio de direcciones del programa se cargó comenzando en la dirección 3000 (y no 0, como piensa el programa), el cargador reescribiría la instrucción para compensar cada dirección en 3000 (por ejemplo, `movl 4000, %eax`). De esta forma se consigue una reubicación estática sencilla del espacio de direcciones del proceso.

Sin embargo, la reubicación estática plantea numerosos problemas. En primer lugar y lo más importante, no proporciona protección, ya que los procesos pueden generar direcciones incorrectas y, por lo tanto, acceder ilegalmente a la memoria de otros procesos o incluso del sistema operativo; en general, es probable que se necesite soporte de hardware para una verdadera protección [WL+93]. Otro aspecto negativo es que una vez colocado, es difícil reubicar posteriormente un espacio de direcciones en otra ubicación [M65].

espacio de direcciones en cualquier lugar que queramos en la memoria física, y hacerlo asegurándonos de que el proceso solo pueda acceder a su propio espacio de direcciones.

En esta configuración, cada programa se escribe y compila como si estuviera cargado en la dirección cero. Sin embargo, cuando un programa comienza a ejecutarse, el sistema operativo decide en qué parte de la memoria física debe cargarse y establece el registro base en ese valor. En el ejemplo anterior, el sistema operativo decide cargar el proceso en la dirección física de 32 KB y, por lo tanto, establece el registro base en este valor.

Cosas interesantes empiezan a suceder cuando el proceso está en marcha. Ahora, cuando el proceso genera cualquier referencia de memoria, el procesador la traduce de la siguiente manera:

$$\text{dirección física} = \text{dirección virtual} + \text{base}$$

Cada referencia de memoria generada por el proceso es una dirección virtual; el hardware, a su vez, agrega el contenido del registro base a esta dirección y el resultado es una dirección física que se puede emitir al sistema de memoria.

Para entender esto mejor, analicemos lo que sucede cuando se ejecuta una sola instrucción. Específicamente, veamos una instrucción de nuestra secuencia anterior:

`128: movimiento 0x0(%ebx), %eax`

El contador de programa (PC) está configurado en 128; cuando el hardware necesita recuperar esta instrucción, primero agrega el valor al valor del registro base de 32 KB (32768) para obtener una dirección física de 32896; Luego, el hardware recupera la instrucción de esa dirección física. A continuación, el procesador comienza a ejecutar la instrucción. En algún momento, el proceso emite

**CONSEJO: REUBICACIÓN DINÁMICA BASADA EN HARDWARE**

Con la reubicación dinámica, un poco de hardware ayuda mucho. Es decir, se utiliza un registro base para transformar direcciones virtuales (generadas por el programa) en direcciones físicas. Un registro de límites (o límite) garantiza que dichas direcciones estén dentro de los límites del espacio de direcciones. Juntos proporcionan una virtualización de la memoria simple y eficiente.

la carga desde la dirección virtual es de 15 KB, que el procesador toma y vuelve a agregar al registro base (32 KB), obteniendo la dirección física final de 47 KB y por tanto el contenido deseado.

Transformar una dirección virtual en una dirección física es exactamente la técnica a la que nos referimos como traducción de direcciones; es decir, el hardware toma una dirección virtual a la que el proceso cree que hace referencia y la transforma en una dirección física que es donde realmente residen los datos. Debido a que esta reubicación de la dirección ocurre en tiempo de ejecución, y debido a que podemos mover espacios de direcciones incluso después de que el proceso haya comenzado a ejecutarse, la técnica a menudo se denomina reubicación dinámica [M65].

Ahora quizás te preguntes: ¿qué pasó con ese registro de límites? Después de todo, ¿no es éste el enfoque de base y límites? De hecho, lo es. Como habrás adivinado, el registro de límites está ahí para ayudar con la protección. Específicamente, el procesador primero verificará que la referencia de memoria esté dentro de los límites para asegurarse de que sea legal; En el ejemplo simple anterior, el registro de límites siempre se establecería en 16 KB. Si un proceso genera una dirección virtual mayor que los límites, o una que es negativa, la CPU generará una excepción y es probable que el proceso finalice.

El objetivo de los límites es, por tanto, garantizar que todas las direcciones generadas por el proceso sean legales y estén dentro de los "límites" del proceso.

Debemos tener en cuenta que los registros de base y límites son estructuras de hardware guardadas en el chip (un par por CPU). A veces la gente llama unidad de gestión de memoria (MMU) a la parte del procesador que ayuda con la traducción de direcciones ; A medida que desarrollemos técnicas de gestión de memoria más sofisticadas, añadiremos más circuitos a la MMU.

Un pequeño comentario aparte sobre los registros enlazados, que se pueden definir de dos maneras. De una manera (como arriba), mantiene el tamaño del espacio de direcciones y, por lo tanto, el hardware verifica la dirección virtual primero antes de agregar la base. En la segunda forma, mantiene la dirección física del final del espacio de direcciones y, por lo tanto, el hardware primero agrega la base y luego se asegura de que la dirección esté dentro de los límites. Ambos métodos son lógicamente equivalentes; Por simplicidad, normalmente asumiremos el primer método.

#### Traducciones de ejemplo Para

comprender la traducción de direcciones mediante bases y límites con más detalle, veamos un ejemplo. Imagine que un proceso con un espacio de direcciones de 4 KB (sí, irrealmente pequeño) se ha cargado en una dirección física de 16 KB. A continuación se muestran los resultados de varias traducciones de direcciones:

Dirección virtual 0 →	Dirección física
	16 KB
1 KB → 3000	17 KB
→	19384
4400 → Fallo (fuera de límites)	

Como puede ver en el ejemplo, le resultará fácil simplemente agregar el dirección base a la dirección virtual (que con razón puede verse como una desplazado en el espacio de direcciones) para obtener la dirección física resultante. Solo si la dirección virtual es "demasiado grande" o negativa, el resultado será una falla, provocando que se genere una excepción.

## 15.4 Soporte de hardware: resumen

Resumamos ahora el soporte que necesitamos del hardware (también consulte la Figura 15.3, página 9). Primero, como se analizó en el capítulo sobre virtualización de CPU, necesitamos dos modos de CPU diferentes. El sistema operativo se ejecuta en privilegios. modo (o modo kernel), donde tiene acceso a toda la máquina; Las aplicaciones se ejecutan en modo de usuario, donde están limitadas en lo que pueden hacer. A un solo bit, tal vez almacenado en algún tipo de palabra de estado del procesador, indica en qué modo se está ejecutando actualmente la CPU; sobre ciertos especiales ocasiones (por ejemplo, una llamada al sistema o algún otro tipo de excepción o interrupción), la CPU cambia de modo.

El hardware también debe proporcionar los propios registros de base y límites ; Por tanto, cada CPU tiene un par adicional de registros, que forman parte de la unidad de gestión de memoria (MMU) de la CPU. Cuando un programa de usuario se está ejecutando, el hardware traducirá cada dirección sumando el valor base a la dirección virtual generada por el programa de usuario. El hardware debe También podrá comprobar si la dirección es válida, lo cual se logra utilizando el registro de límites y algunos circuitos dentro de la CPU.

El hardware debe proporcionar instrucciones especiales para modificar la base. y límites de registros, lo que permite al sistema operativo cambiarlos cuando sean diferentes. se ejecutan los procesos. Estas instrucciones son privilegiadas; sólo en modo kernel (o privilegiado) se pueden modificar los registros. Imagínese los estragos que un usuario El proceso podría arruinarse<sup>1</sup> si pudiera cambiar arbitrariamente el registro base mientras

<sup>1</sup>¿Hay algo más que "estragos" que se pueda "causar"? [W17]

### APARTE: ESTRUCTURA DE DATOS : LA LISTA GRATUITA

El sistema operativo debe rastrear qué partes de la memoria libre no están en uso, para Ser capaz de asignar memoria a los procesos. Muchas estructuras de datos diferentes Por supuesto, puede utilizarse para tal tarea; el más simple (que asumiremos aquí) es una lista gratuita, que simplemente es una lista de los rangos de la física memorias que no están actualmente en uso.

Requisitos de hardware	Notas
Modo privilegiado	Necesario para evitar que los procesos en modo de usuario ejecuten operaciones privilegiadas
Registros de base/límites	Necesita un par de registros por CPU para admitir la traducción de direcciones y las comprobaciones de límites
Capacidad para traducir direcciones virtuales	Circuitos para realizar traducciones y verificar límites; en este caso, y comprobar si está dentro de los límites
Instrucciones privilegiadas para valores de base/límites de actualización	bastante simple El sistema operativo debe poder establecer estos
Las instrucciones privilegiadas para registrar el sistema operativo	deben poder indicarle al hardware qué manejadores de excepciones
capacidad para plantear excepciones.	código para ejecutar si ocurre una excepción
	Cuando los procesos intentan acceder a instrucciones privilegiadas o a memoria fuera de los límites

Figura 15.3: Reubicación dinámica: requisitos de hardware

correr. ¡Imagínatelo! Y luego rápidamente elimina esos pensamientos oscuros de tu mente, ya que son la materia espantosa de la que están hechas las pesadillas.

Finalmente, la CPU debe poder generar excepciones en situaciones donde un programa de usuario intenta acceder a la memoria ilegalmente (con una dirección que está "fuera de límites"); en este caso, la CPU debería dejar de ejecutar el programa de usuario y organizar la ejecución del controlador de excepciones "fuera de límites" del sistema operativo . El controlador del sistema operativo puede entonces descubrir cómo reaccionar, en este caso probablemente finalizando el proceso. De manera similar, si un programa de usuario intenta cambiar los valores de los registros de base y límites (privilegiados), la CPU debería generar una excepción y ejecutar el controlador "inténtate ejecutar una operación privilegiada mientras estaba en modo de usuario". La CPU también debe proporcionar un método para informarle de la ubicación de estos controladores; Por tanto, se necesitan algunas instrucciones privilegiadas más.

## 15.5 Problemas del sistema operativo

Así como el hardware proporciona nuevas características para soportar la reubicación dinámica, el sistema operativo ahora tiene nuevos problemas que debe manejar; la combinación de soporte de hardware y gestión del sistema operativo conduce a la implementación de una memoria virtual simple. Específicamente, hay algunas coyunturas críticas en las que el sistema operativo debe intervenir para implementar nuestra versión básica y con límites de memoria virtual.

En primer lugar, el sistema operativo debe actuar cuando se crea un proceso y encontrar espacio para su dirección en la memoria. Afortunadamente, dadas nuestras suposiciones de que cada espacio de direcciones es (a) más pequeño que el tamaño de la memoria física y (b) del mismo tamaño, esto es bastante fácil para el sistema operativo; simplemente puede ver la memoria física como una serie de ranuras y realizar un seguimiento de si cada una está libre o en uso. Cuando se crea un nuevo proceso, el sistema operativo tendrá que buscar en una estructura de datos (a menudo llamada lista libre) para encontrar espacio para el nuevo espacio de direcciones y luego marcarlo como usado. Con espacios de direcciones de tamaño variable, la vida es más complicada, pero dejaremos esa preocupación para capítulos futuros.

Requisitos del sistema operativo	Notas
Gestión de la memoria	Necesidad de asignar memoria para nuevos procesos; Recuperar memoria de procesos terminados; Generalmente administre la memoria a través de la lista libre
Gestión de bases/límites. Debe establecer bases/límites correctamente al cambiar de contexto.	
Manejo de excepciones	Código para ejecutar cuando surjan excepciones; La acción probable es poner fin al proceso infractor.

Figura 15.4: Reubicación dinámica: Responsabilidades del sistema operativo

Veamos un ejemplo. En la Figura 15.2 (página 5), puede ver el sistema operativo utilizando la primera ranura de memoria física para sí mismo y que se ha reubicado. Siga el proceso del ejemplo anterior en la ranura comenzando en la dirección de memoria física de 32 KB. Los otros dos slots están libres (16 KB-32 KB y 48 KB-64 KB); por tanto, la lista libre debería constar de estas dos entradas.

En segundo lugar, el sistema operativo debe realizar algún trabajo cuando finaliza un proceso (es decir, cuando sale con gracia, o es asesinado a la fuerza porque se portó mal), recuperar toda su memoria para usarla en otros procesos o en el sistema operativo. Al finalizar un proceso, el sistema operativo vuelve a poner su memoria en el lugar libre. Lista y limpia cualquier estructura de datos asociada según sea necesario.

En tercer lugar, el sistema operativo también debe realizar algunos pasos adicionales cuando un contexto se produce el cambio. Sólo hay un par de registros de base y límites en cada CPU, después de todo, y sus valores difieren para cada programa en ejecución, ya que cada programa se carga en una dirección física diferente en la memoria. Así, el sistema operativo debe guardar y restaurar el par de base y límites cuando cambia entre procesos. Específicamente, cuando el sistema operativo decide detener la ejecución de un proceso, debe guardar los valores de los registros base y de límites en la memoria. en alguna estructura por proceso, como la estructura de proceso o el proceso bloque de control (PCB). De manera similar, cuando el sistema operativo reanuda un proceso en ejecución (o lo ejecuta la primera vez), debe establecer los valores de la base y los límites en la CPU a los valores correctos para este proceso.

Debemos tener en cuenta que cuando un proceso se detiene (es decir, no se ejecuta), es posible que el sistema operativo mueva un espacio de direcciones de una ubicación en la memoria a otra con bastante facilidad. Para mover el espacio de direcciones de un proceso, el sistema operativo primero desprograma el proceso; luego, el sistema operativo copia el espacio de direcciones de la ubicación actual a la nueva ubicación; Finalmente, el sistema operativo actualiza los archivos guardados. registro base (en la estructura del proceso) para señalar la nueva ubicación. Cuando el proceso se reanuda, se restaura su (nuevo) registro base y comienza ejecutándose de nuevo, sin darse cuenta de que sus instrucciones y datos ahora están en un lugar completamente nuevo en la memoria.

Cuarto, el sistema operativo debe proporcionar controladores de excepciones o funciones que se llamo, como se discutió anteriormente; el sistema operativo instala estos controladores en el momento del arranque (a través de instrucciones privilegiadas). Por ejemplo, si un proceso intenta acceder a la memoria fuera de sus límites, la CPU generará una excepción; el sistema operativo debe ser preparados para tomar medidas cuando surja tal excepción. La reacción común del sistema operativo será de hostilidad: probablemente pondrá fin a la infracción. proceso. El sistema operativo debe proteger mucho la máquina que está ejecutando, y por lo tanto no le agrada que un proceso intente acceder a la memoria o

OS @ boot (modo kernel)	Hardware	(Aún no hay programa)
inicializa la tabla de capturas	recordar direcciones de... controlador de llamadas al sistema controlador de	
	temporizador controlador de acceso a memoria ilegal controlador de instrucciones ilegal	
iniciar el temporizador de interrupción		iniciar el temporizador; interrumpir después de X ms
inicializar tabla de procesos		
inicializar lista libre		

Figura 15.5: Ejecución directa limitada (reubicación dinámica) en el arranque

ejecutar instrucciones que no debería. Adiós, proceso de mal comportamiento; Ha sido un placer conocerte.

Las figuras 15.5 y 15.6 (página 12) ilustran gran parte de la interacción hardware/OS en una línea de tiempo. La primera figura muestra lo que hace el sistema operativo en el momento del arranque para preparar la máquina para su uso, y la segunda muestra lo que sucede cuando un proceso (Proceso A) comienza a ejecutarse; observe cómo el hardware maneja sus traducciones de memoria sin intervención del sistema operativo. En algún momento (en la mitad de la segunda figura), se produce una interrupción del temporizador y el sistema operativo cambia al Proceso B, que ejecuta una "carga incorrecta" (a una dirección de memoria ilegal); en ese punto, el sistema operativo debe intervenir, finalizando el proceso y limpiando liberando la memoria de B y eliminando su entrada de la tabla de procesos. Como puede verse en las figuras, todavía seguimos el enfoque básico de ejecución directa limitada. En la mayoría de los casos, el sistema operativo simplemente configura el hardware adecuadamente y permite que el proceso se ejecute directamente en la CPU; sólo cuando el proceso se comporta mal el sistema operativo tiene que intervenir.

## 15.6 Resumen

En este capítulo, hemos ampliado el concepto de ejecución directa limitada con un mecanismo específico utilizado en la memoria virtual, conocido como traducción de direcciones. Con la traducción de direcciones, el sistema operativo puede controlar todos y cada uno de los accesos a la memoria desde un proceso, asegurando que los accesos permanezcan dentro de los límites del espacio de direcciones. La clave para la eficiencia de esta técnica es el soporte de hardware, que realiza la traducción rápidamente para cada acceso, convirtiendo las direcciones virtuales (la vista de la memoria del proceso) en físicas (la vista real). Todo esto se realiza de forma transparente al proceso que se ha reubicado; el proceso no tiene idea de que se están traduciendo las referencias de su memoria, lo que crea una maravillosa ilusión.

También hemos visto una forma particular de virtualización, conocida como base y límites o reubicación dinámica. La virtualización de base y límites es bastante eficiente, ya que sólo se requiere un poco más de lógica de hardware para agregar un

OS @ run (modo kernel)	Hardware	Programa (modo usuario)
Para iniciar el proceso A: asigne una entrada en la tabla de procesos, asigne memoria para el conjunto de procesos, registros base/enlazados, retorno desde la trampa (en A)	restaurar los registros de A pasar al modo de usuario saltar a la PC (inicial) de A	El proceso A se ejecuta Obtener instrucciones
	traducir dirección virtual realizar búsqueda	Ejecutar instrucción
	si es explícito cargar/almacenar: asegúrese de que la dirección sea legal traducir la dirección virtual realizar carga/almacenamiento	
		(Un corre...)
	Interrupción del temporizador pasar al modo kernel saltar al controlador	
Manejar el temporizador decidir: detener A, ejecutar B llamar a la rutina switch() guardar regs(A) en proc-struct(A) (incluyendo base/límites) restaurar regs(B) desde proc- struct(B) (incluyendo base/ límites) regreso de la trampa (a B)	restaurar los registros de B pasar al modo de usuario saltar a la PC de B	El proceso B se ejecuta Ejecutar mala carga
	La carga está fuera de los límites; pasar al modo kernel saltar al controlador de trampas	
Manejar la trampa, decidir matar el proceso B, desasignar la memoria de B, liberar la entrada de B en la tabla de procesos		

Figura 15.6: Ejecución directa limitada (reubicación dinámica) en tiempo de ejecución

regístrese base en la dirección virtual y verifique que la dirección generada por el proceso esté dentro de los límites. Base-and-bounds también ofrece protección; el sistema operativo y el hardware se combinan para garantizar que ningún proceso pueda generar referencias de memoria fuera de su propio espacio de direcciones. La protección es sin duda uno de los objetivos más importantes del sistema operativo; sin él, el sistema operativo no podría controlar la máquina (si los procesos tuvieran libertad para sobrescribir la memoria, fácilmente podrían hacer cosas desagradables como sobrescribir la tabla de captura y apoderarse del sistema).

Desafortunadamente, esta sencilla técnica de reubicación dinámica tiene sus inefficiencias. Por ejemplo, como puede ver en la Figura 15.2 (página 5), el proceso reubicado utiliza memoria física de 32 KB a 48 KB; sin embargo, debido a que la pila y el montón de procesos no son demasiado grandes, todo el espacio entre los dos simplemente se desperdicia. Este tipo de desperdicio generalmente se denomina fragmentación interna, ya que el espacio dentro de la unidad asignada no se utiliza en su totalidad (es decir, se fragmenta) y, por lo tanto, se desperdicia. En nuestro enfoque actual, aunque podría haber suficiente memoria física para más procesos, actualmente estamos restringidos a colocar un espacio de direcciones en una ranura de tamaño fijo y, por lo tanto, puede surgir fragmentación interna<sup>2</sup>. Por tanto, vamos a necesitar maquinaria más sofisticada para intentar utilizar mejor la memoria física y evitar la fragmentación interna. Nuestro primer intento será una ligera generalización de la base y los límites conocida como segmentación, que discutiremos a continuación.

---

<sup>2</sup>Una solución diferente podría, en cambio, colocar una pila de tamaño fijo dentro del espacio de direcciones, justo debajo de la región del código, y un montón creciente debajo de ella. Sin embargo, esto limita la flexibilidad al hacer que la recursividad y las llamadas a funciones profundamente anidadas sean un desafío y, por lo tanto, es algo que esperamos evitar.

## Referencias

[M65] "Sobre la reubicación dinámica de programas" por WC McGee. IBM Systems Journal, volumen 4:3, 1965, páginas 184–199. Este artículo es un buen resumen de los primeros trabajos sobre reubicación dinámica, así como algunos conceptos básicos sobre reubicación estática.

[P90] "Reubicación del cargador para archivos ejecutables .EXE de MS-DOS" por Kenneth DA Pillay. Archivo de microprocesadores y microsistemas, volumen 14:7 (septiembre de 1990). Un ejemplo de cargador reubicado para MS-DOS. No es el primero, sino sólo un ejemplo relativamente moderno de cómo funciona un sistema de este tipo.

[SS74] "La protección de la información en los sistemas informáticos" por J. Saltzer y M. Schroeder. CACM, julio de 1974. De este artículo: "Los conceptos de registro de base y límite y descriptores interpretados por hardware aparecieron, aparentemente de forma independiente, entre 1957 y 1959 en tres proyectos con objetivos diversos. En el MIT, McCarthy sugirió la idea de base y límite como parte del sistema de protección de la memoria necesario para hacer factible el tiempo compartido. IBM desarrolló de forma independiente el registro de base y límite como un mecanismo para permitir una multiprogramación confiable del sistema informático Stretch (7030). En Burroughs, R. Barton sugirió que los descriptores interpretados por hardware proporcionarían soporte directo para las reglas de alcance de nombres de lenguajes de nivel superior en el sistema informático B5000". Encontramos esta cita en las interesantes páginas de historia de Mark Smotherman [S04]; véalos para más información.

[S04] "Soporte de llamadas al sistema" por Mark Smotherman. Mayo de 2004. [people.cs.clemson.edu/~mark/syscall.html](http://people.cs.clemson.edu/~mark/syscall.html). Una buena historia de soporte de llamadas al sistema. Smotherman también ha recopilado algo de historia temprana sobre elementos como interrupciones y otros aspectos divertidos de la historia de la informática. Consulte sus páginas web para obtener más detalles.

[WL+93] "Aislamiento de fallas basado en software eficiente" por Robert Wahbe, Steven Lucco, Thomas E. Anderson, Susan L. Graham. SOSP '93. Un excelente artículo sobre cómo se puede utilizar la compatibilidad con el compilador para vincular referencias de memoria desde un programa, sin compatibilidad con el hardware. El artículo despertó un renovado interés en las técnicas de software para el aislamiento de referencias de memoria.

[W17] Respuesta a la nota al pie: "¿Hay algo más que estragos que se pueda causar?" por Waciuma Wanjohi. Octubre de 2017. Sorprendentemente, este lector emprendedor encontró la respuesta a través de la herramienta de visualización Ngram de Google (disponible en la siguiente URL: <http://books.google.com/ngrams>).

La respuesta, gracias al Sr. Wanjohi: "Sólo a partir de 1970 aproximadamente, 'sembrar el caos' ha sido más popular que 'sembrar venganza'. En el siglo XIX, la palabra causar casi siempre iba seguida de 'su/su venganza'. Aparentemente, cuando haces destrozos, no tramas nada bueno, pero al menos los destrozadores ahora tienen algunas opciones.

## Tarea (Simulación)

El programa relocation.py le permite ver cómo se realizan las traducciones de direcciones en un sistema con registros de base y límites. Consulte el archivo LÉAME para obtener más detalles.

## Preguntas

1. Ejecute con las semillas 1, 2 y 3 y calcule si cada dirección virtual generada por el proceso está dentro o fuera de los límites. Si está dentro de los límites, calcule la traducción.
2. Ejecute con estos indicadores: -s 0 -n 10. ¿A qué valor ha configurado -l (el registro de límites) para garantizar que todas las direcciones virtuales generadas estén dentro de los límites?
3. Ejecute con estos indicadores: -s 1 -n 10 -l 100. ¿Cuál es el valor máximo que se puede establecer en esa base, de modo que el espacio de direcciones aún queda en la memoria física en su totalidad?
4. Ejecute algunos de los mismos problemas anteriores, pero con espacios de direcciones (-a) y memorias físicas (-p) más grandes.
5. ¿Qué fracción de direcciones virtuales generadas aleatoriamente son válidas, en función del valor del registro de límites? Haga un gráfico ejecutando con diferentes semillas aleatorias, con valores límite que van desde 0 hasta el tamaño máximo del espacio de direcciones.

## Segmentación

Hasta ahora hemos estado poniendo en memoria todo el espacio de direcciones de cada proceso. Con los registros de base y límites, el sistema operativo puede reubicar fácilmente procesos en diferentes partes de la memoria física. Sin embargo, es posible que hayas notado algo interesante acerca de estos espacios de direcciones nuestros: hay una gran porción de espacio "libre" justo en el medio, entre la pila y el montón.

Como se puede imaginar en la Figura 16.1, aunque el proceso no utiliza el espacio entre la pila y el montón, todavía ocupa memoria física cuando reubicamos todo el espacio de direcciones en algún lugar de la memoria física; por lo tanto, el enfoque simple de utilizar un par de registros de base y límites para virtualizar la memoria es un desperdicio. También dificulta bastante la ejecución de un programa cuando todo el espacio de direcciones no cabe en la memoria; por tanto, la base y los límites no son tan flexibles como nos gustaría. Y así:

### EL CRUX: CÓMO SOPORTAR UN GRAN ESPACIO DE DIRECCIONES

¿Cómo admitimos un gran espacio de direcciones con (potencialmente) mucho espacio libre entre la pila y el montón? Tenga en cuenta que en nuestros ejemplos, con espacios de direcciones pequeños (ficticios), el desperdicio no parece tan grave. Imagine, sin embargo, un espacio de direcciones de 32 bits (4 GB de tamaño); un programa típico sólo utilizará megabytes de memoria, pero aun así exigirá que todo el espacio de direcciones resida en la memoria.

### 16.1 Segmentación: base/límites generalizados

Para solucionar este problema nació una idea y se llama segmentación. Es una idea bastante antigua, que se remonta al menos a principios de los años 1960 [H61, G62]. La idea es simple: en lugar de tener solo un par de bases y límites en nuestra MMU, ¿por qué no tener un par de bases y límites por segmento lógico del espacio de direcciones? Un segmento es solo una porción contigua del espacio de direcciones de una longitud particular, y en nuestro canónico

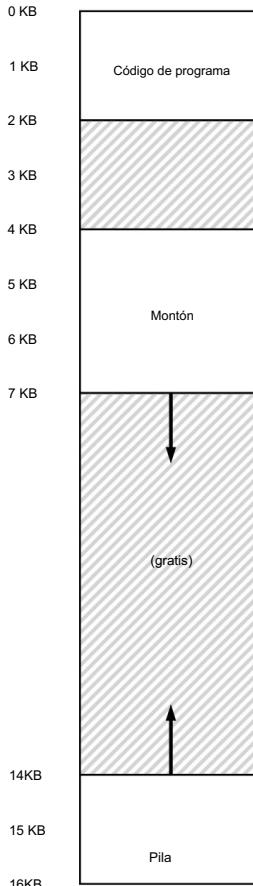


Figura 16.1: Un espacio de direcciones (nuevamente)

espacio de direcciones, tenemos tres segmentos lógicamente diferentes: código, pila y montón. Lo que la segmentación permite al SO es colocar cada uno de esos segmentos en diferentes partes de la memoria física, y así evitar llenar la memoria física con espacio de direcciones virtuales no utilizado.

Veamos un ejemplo. Supongamos que queremos colocar el espacio de direcciones de la Figura 16.1 en la memoria física. Con un par de bases y límites por segmento, podemos colocar cada segmento de forma independiente en la memoria física. Por ejemplo, consulte la Figura 16.2 (página 3); allí ves una memoria física de 64 KB con esos tres segmentos (y 16 KB reservados para el sistema operativo).

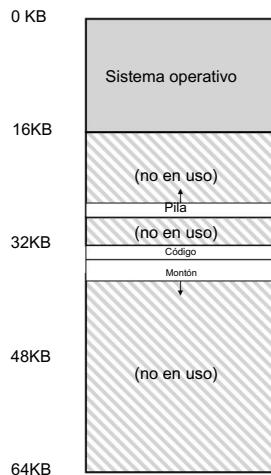


Figura 16.2: Colocación de segmentos en la memoria física

Como puede ver en el diagrama, solo a la memoria usada se le asigna espacio en la memoria física y, por lo tanto, se pueden acomodar grandes espacios de direcciones con grandes cantidades de espacio de direcciones no utilizado (que a veces llamamos espacios de direcciones dispersos).

La estructura de hardware en nuestra MMU necesaria para soportar la segmentación es justo lo que cabría esperar: en este caso, un conjunto de tres pares de registros de base y límites. La Figura 16.3 a continuación muestra los valores de registro para el ejemplo anterior; cada registro de límites tiene el tamaño de un segmento.

Tamaño base del segmento	
Código	32K 2K
Montón	34K 3K
Pila	28K 2K

Figura 16.3: Valores de registro de segmento

Puede ver en la figura que el segmento de código se coloca en la dirección física de 32 KB y tiene un tamaño de 2 KB y el segmento de montón se coloca en 34 KB y tiene un tamaño de 3 KB. El segmento de tamaño aquí es exactamente el mismo que el del registro de límites introducido anteriormente; le dice al hardware exactamente cuántos bytes son válidos en este segmento (y, por lo tanto, permite al hardware determinar cuándo un programa ha realizado un acceso ilegal fuera de esos límites).

Hagamos una traducción de ejemplo, usando el espacio de direcciones de la Figura 16.1. Supongamos que se hace una referencia a la dirección virtual 100 (que se encuentra en el segmento de código, como puede ver visualmente en la Figura 16.1, página 2). Cuando la referencia

## APARTE: LA FALLA DE SEGMENTACIÓN El término

falla o violación de segmentación surge de un acceso a la memoria en una máquina segmentada a una dirección ilegal. Curiosamente, el término persiste, incluso en máquinas que no admiten la segmentación en absoluto. O no con tanto humor, si no puedes entender por qué tu código sigue fallando.

ocurre una comparación (por ejemplo, en una búsqueda de instrucciones), el hardware agregará el valor base al desplazamiento en este segmento (100 en este caso) para llegar a la dirección física deseada:  $100 + 32\text{ KB}$ , o 32868. Luego verificará que la dirección está dentro de los límites (100 es menos de 2 KB), determine que así es y emita la referencia a la dirección de memoria física 32868.

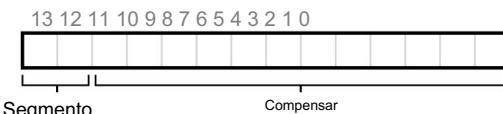
Ahora veamos una dirección en el montón, la dirección virtual 4200 (consulte nuevamente la Figura 16.1). Si simplemente agregamos la dirección virtual 4200 a la base del montón (34 KB), obtenemos una dirección física de 39016, que no es la dirección física correcta. Lo primero que debemos hacer es extraer el desplazamiento en el montón, es decir, a qué bytes de este segmento se refiere la dirección. Debido a que el montón comienza en la dirección virtual 4 KB (4096), el desplazamiento de 4200 es en realidad 4200 menos 4096, o 104. Luego tomamos este desplazamiento (104) y lo agregamos a la dirección física del registro base (34 K) para obtener el resultado deseado. : 34920.

¿Qué pasaría si intentáramos hacer referencia a una dirección ilegal (es decir, una dirección virtual de 7 KB o más), que está más allá del final del montón? Puede imaginar lo que sucederá: el hardware detecta que la dirección está fuera de los límites, entra en el sistema operativo y probablemente provoca la finalización del proceso infractor. Y ahora ya conoces el origen del famoso término que todos los programadores de C aprenden a temer: la violación de segmentación o fallo de segmentación.

## 16.2 ¿A qué segmento nos referimos?

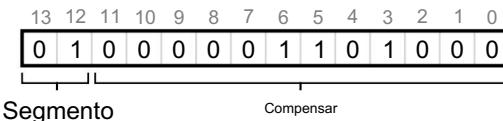
El hardware utiliza registros de segmento durante la traducción. ¿Cómo sabe el desplazamiento en un segmento y a qué segmento se refiere una dirección?

Un enfoque común, a veces denominado enfoque explícito , es dividir el espacio de direcciones en segmentos basados en los pocos bits superiores de la dirección virtual; esta técnica se utilizó en el sistema VAX/VMS [LL82]. En nuestro ejemplo anterior, tenemos tres segmentos; por tanto, necesitamos dos bits para realizar nuestra tarea. Si usamos los dos bits superiores de nuestra dirección virtual de 14 bits para seleccionar el segmento, nuestra dirección virtual se verá así:



En nuestro ejemplo, entonces, si los dos bits superiores son 00, el hardware sabe que la dirección virtual está en el segmento de código y, por lo tanto, utiliza el par de base y límites del código para reubicar la dirección en la ubicación física correcta. Si los dos bits superiores son 01, el hardware sabe que la dirección está en el montón,

y por lo tanto utiliza la base y los límites del montón. Tomemos nuestro ejemplo de dirección virtual del montón anterior (4200) y tradujémoslo, solo para asegurarnos de que quede claro. La dirección virtual 4200, en formato binario, se puede ver aquí:



Como puede ver en la imagen, los dos bits superiores (01) le indican al hardware a qué segmento nos referimos. Los 12 bits inferiores son el desplazamiento en el segmento: 0000 0110 1000, o hexadecimal 0x068, o 104 en decimal. Por lo tanto, el hardware simplemente toma los primeros dos bits para determinar qué registro de segmento usar y luego toma los siguientes 12 bits como desplazamiento dentro del segmento.

Al agregar el registro base al desplazamiento, el hardware llega a la dirección física final. Tenga en cuenta que el desplazamiento también facilita la verificación de los límites: simplemente podemos verificar si el desplazamiento es menor que los límites; si no, la dirección es ilegal. Por lo tanto, si la base y los límites fueran matrices (con una entrada por segmento), el hardware haría algo como esto para obtener la dirección física deseada:

- 1 // obtiene los 2 bits superiores de VA de 14 bits
- 2 Segmento = (VirtualAddress & SEG\_MASK) >> SEG\_SHIFT 3 // ahora obtenemos compensación
- 4 Desplazamiento = Dirección virtual y OFFSET\_MASK 5 si  
(Desplazamiento >= Límites[Segmento])
- 6 RaiseException(PROTECTION\_FAULT) 7 más
- 8 PhysAddr = Base[Segmento] + Desplazamiento
- 9 Registro = AccessMemory(PhysAddr)

En nuestro ejemplo en ejecución, podemos completar valores para las constantes anteriores. Específicamente, SEG-MASK se establecería en 0x3000, SEG SHIFT en 12 y MÁSCARA DE DESPLAZAMIENTO a 0xFFFF.

También habrás notado que cuando usamos los dos bits superiores y solo tenemos tres segmentos (código, montón, pila), un segmento del espacio de direcciones no se utiliza. Para utilizar completamente el espacio de direcciones virtuales (y evitar un segmento no utilizado), algunos sistemas colocan el código en el mismo segmento que el montón y, por lo tanto, usan solo un bit para seleccionar qué segmento usar [Li-82].

Otro problema con el uso de tantos bits superiores para seleccionar un segmento es que limita el uso del espacio de direcciones virtuales. Específicamente, cada segmento está limitado a un tamaño máximo, que en nuestro ejemplo es 4 KB (usar los dos bits superiores para elegir segmentos implica que el espacio de direcciones de 16 KB se divide en cuatro partes, o 4 KB en este ejemplo). Si un programa en ejecución desea hacer crecer un segmento (por ejemplo, el montón o la pila) más allá de ese máximo, el programa no tiene suerte.

Hay otras formas para que el hardware determine en qué segmento se encuentra una dirección particular. En el enfoque implícito, el hardware determina

mina el segmento al notar cómo se formó la dirección. Si, por ejemplo, la dirección se generó a partir del contador del programa (es decir, fue una instrucción de búsqueda), entonces la dirección está dentro del segmento de código; si el dirección se basa fuera de la pila o del puntero base, debe estar en la pila. La segmento; cualquier otra dirección debe estar en el montón.

### 16.3 ¿Qué pasa con la pila?

Hasta ahora, hemos omitido un componente importante del espacio de direcciones: la pila. La pila se ha reubicado en la dirección física 28 KB en el diagrama anterior, pero con una diferencia crítica: crece hacia atrás (es decir, hacia direcciones inferiores). En la memoria física, "comienza" en 28 KB<sup>1</sup> y vuelve a crecer a 26 KB, correspondiente a direcciones virtuales de 16 KB a 14 KB; la traducción debe proceder de manera diferente.

Lo primero que necesitamos es un poco de soporte de hardware adicional. En lugar de solo valores base y límites, el hardware también necesita saber en qué dirección el segmento crece (un poco, por ejemplo, que se establece en 1 cuando el segmento crece en la dirección positiva y 0 en la dirección negativa). Nuestra visión actualizada de Lo que sigue el hardware se ve en la Figura 16.4:

¿El tamaño base del segmento (máximo 4K) se vuelve positivo?	
Código	00 32K 2K Montón
01	34K 3K
Pila	11 28K 2K
	1
	1
	0

Figura 16.4: Registros de segmentos (con soporte de crecimiento negativo)

Una vez que el hardware comprende que los segmentos pueden crecer en dirección negativa, el hardware ahora debe traducir dichas direcciones virtuales. ligeramente diferente. Tomemos un ejemplo de dirección virtual de pila y tradúzcalo para comprender el proceso.

En este ejemplo, supongamos que deseamos acceder a la dirección virtual de 15 KB, que debe asignarse a la dirección física 27 KB. Nuestra dirección virtual, en binario formulario, por lo tanto se ve así: 11 1100 0000 0000 (hexadecimal 0x3C00). El hardware utiliza los dos bits superiores (11) para designar el segmento, pero luego estamos quedó con un desplazamiento de 3 KB. Para obtener el desplazamiento negativo correcto, debemos restar el tamaño máximo del segmento de 3 KB: en este ejemplo, un segmento puede tener 4 KB y, por lo tanto, el desplazamiento negativo correcto es 3 KB menos 4 KB. lo que equivale a -1 KB. Simplemente agregamos el desplazamiento negativo (-1KB) a la base (28KB) para llegar a la dirección física correcta: 27KB. Los límites verifican se puede calcular asegurándose de que el valor absoluto del desplazamiento negativo sea menor o igual que el tamaño actual del segmento (en este caso, 2 KB).

---

<sup>1</sup>Aunque decimos, por simplicidad, que la pila "comienza" en 28 KB, este valor es en realidad el byte justo debajo de la ubicación de la región de crecimiento atrasado; el primer byte válido es en realidad 28 KB menos 1. Por el contrario, las regiones de crecimiento hacia adelante comienzan en la dirección del primer byte del segmento. Tomamos este enfoque porque hace que las matemáticas calculen la dirección física. sencillo: la dirección física es solo la base más el desplazamiento negativo.

## 16.4 Soporte para compartir

A medida que crecía el apoyo a la segmentación, los diseñadores de sistemas pronto se dieron cuenta de que podrían lograr nuevos tipos de eficiencias con un poco más de hardware apoyo. En concreto, para ahorrar memoria, a veces es útil compartir ciertos segmentos de memoria entre espacios de direcciones. En particular, código compartir es común y todavía se utiliza en los sistemas actuales.

Para permitir el uso compartido, necesitamos un poco de soporte adicional del hardware, en forma de bits de protección. El soporte básico agrega algunos bits por segmento, indicando si un programa puede leer o escribir un segmento, o tal vez ejecutar código que se encuentra dentro del segmento. Estableciendo un segmento de código a solo lectura, el mismo código se puede compartir entre múltiples procesos, sin preocuparse de dañar el aislamiento; Mientras cada proceso todavía piensa que está accediendo a su propia memoria privada, el sistema operativo comparte memoria en secreto, lo que no puede ser modificado por el proceso y, por tanto, se conserva la ilusión.

Un ejemplo de la información adicional rastreada por el hardware. (y OS) se muestra en la Figura 16.5. Como puede ver, el segmento de código es configurado para leer y ejecutar y, por lo tanto, el mismo segmento físico en la memoria podría asignarse a múltiples espacios de direcciones virtuales.

¿El tamaño base del segmento (máximo 4K) se vuelve positivo?	Protección
Código00 32K 2K Montón01 34K 3K	1 Leer-ejecutar
Pila11 28K 2K	1 Lectura-Escritura 0 Lectura-Escritura

Figura 16.5: Valores de registro de segmento (con protección)

Con bits de protección, el algoritmo de hardware descrito anteriormente también hay que cambiar. Además de comprobar si una dirección virtual es dentro de los límites, el hardware también tiene que comprobar si un acceso en particular está permitido. Si un proceso de usuario intenta escribir en un segmento de solo lectura, o ejecutar desde un segmento no ejecutable, el hardware debería generar una excepción y, por tanto, dejar que el sistema operativo se ocupe del proceso infractor.

## 16.5 Segmentación de grano fino versus segmentación de grano grueso

La mayoría de nuestros ejemplos hasta ahora se han centrado en sistemas con sólo una o pocas segmentos (es decir, código, pila, montón); podemos pensar en esta segmentación de grano grueso, ya que divide el espacio de direcciones en espacios relativamente grandes, trozos gruesos. Sin embargo, algunos de los primeros sistemas (por ejemplo, Multics [CV65,DD68]) eran más flexibles y permitían que los espacios de direcciones consistieran en una gran cantidad de número de segmentos más pequeños, lo que se conoce como segmentación de grano fino.

Para soportar muchos segmentos se requiere aún más soporte de hardware, con una tabla de segmentos de algún tipo almacenada en la memoria. Estas tablas de segmentos normalmente admiten la creación de una gran cantidad de segmentos, y de este modo, permite que un sistema utilice segmentos de formas más flexibles que las que hemos utilizado hasta ahora. hasta ahora discutido. Por ejemplo, las primeras máquinas como la Burroughs B5000 tenía soporte para miles de segmentos y esperaba que un compilador cortara

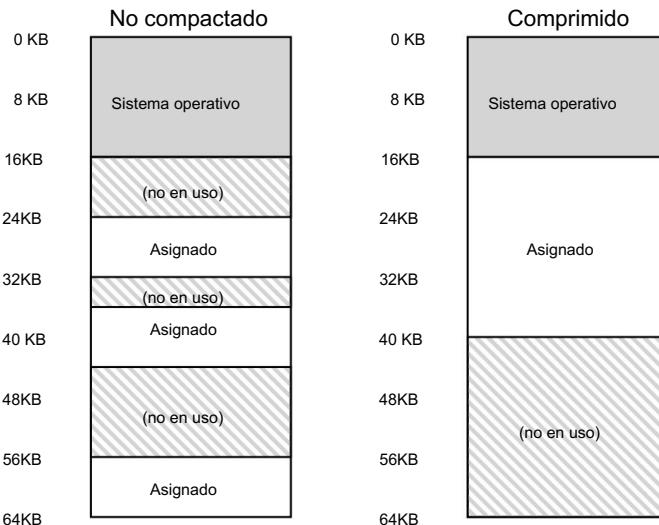


Figura 16.6: Memoria compactada y no compactada

código y datos en segmentos separados que el sistema operativo y el hardware luego admitirían [RK68]. La idea en ese momento era que al tener segmentos detallados, el sistema operativo podría aprender mejor qué segmentos están en uso y cuáles no y, por lo tanto, utilizar la memoria principal de manera más efectiva.

#### 16.6 Soporte del sistema operativo

Ahora debería tener una idea básica de cómo funciona la segmentación. Partes del espacio de direcciones se reubican en la memoria física a medida que se ejecuta el sistema y, por lo tanto, se logra un enorme ahorro de memoria física en relación con nuestro enfoque más simple con un solo par de base/límites para todo el espacio de direcciones. Específicamente, no es necesario asignar todo el espacio no utilizado entre la pila y el montón en la memoria física, lo que nos permite colocar más espacios de direcciones en la memoria física y admitir un espacio de direcciones virtuales grande y escaso por proceso.

Sin embargo, la segmentación plantea una serie de problemas nuevos para el sistema operativo. La primera es antigua: ¿qué debería hacer el sistema operativo en un cambio de contexto? Ya deberías tener una buena idea: los registros de segmento deben guardarse y restaurarse. Claramente, cada proceso tiene su propio espacio de direcciones virtuales y el sistema operativo debe asegurarse de configurar estos registros correctamente antes de permitir que el proceso se ejecute nuevamente.

El segundo es la interacción del sistema operativo cuando los segmentos crecen (o tal vez se reducen). Por ejemplo, un programa puede llamar a `malloc()` para asignar un objeto. En algunos casos, el montón existente podrá atender la solicitud y, por lo tanto,

**CONSEJO: SI EXISTEN 1000 SOLUCIONES , NO HAY UNA GRANDE**

El hecho de que existan tantos algoritmos diferentes para tratar de minimizar la fragmentación externa es indicativo de una verdad subyacente más fuerte: no existe una "mejor" manera de resolver el problema. Por lo tanto, nos conformamos con algo razonable y esperamos que sea lo suficientemente bueno. La única solución real (como veremos en los próximos capítulos) es evitar el problema por completo, no asignando nunca memoria en fragmentos de tamaño variable.

`malloc()` encontrará espacio libre para el objeto y devolverá un puntero a la persona que llama. En otros, sin embargo, es posible que el segmento del montón en sí necesite crecer. En este caso, la biblioteca de asignación de memoria realizará una llamada al sistema para hacer crecer el montón (por ejemplo, la llamada al sistema tradicional UNIX `sbrk()` ). Luego, el sistema operativo proporcionará (normalmente) más espacio, actualizando el registro de tamaño del segmento al nuevo tamaño (más grande) e informando a la biblioteca del éxito; Luego, la biblioteca puede asignar espacio para el nuevo objeto y regresar exitosamente al programa que lo llama. Tenga en cuenta que el sistema operativo podría rechazar la solicitud si no hay más memoria física disponible o si decide que el proceso de llamada ya tiene demasiada.

La última cuestión, y quizás la más importante, es la gestión del espacio libre en la memoria física. Cuando se crea un nuevo espacio de direcciones, el sistema operativo debe poder encontrar espacio en la memoria física para sus segmentos. Anteriormente, asumímos que cada espacio de direcciones tenía el mismo tamaño y, por lo tanto, la memoria física podía considerarse como un conjunto de ranuras donde encajarían los procesos. Ahora, tenemos una cantidad de segmentos por proceso, y cada segmento podría ser diferente. tamaño.

El problema general que surge es que la memoria física rápidamente se llena de pequeños huecos de espacio libre, lo que dificulta la asignación de nuevos segmentos o el crecimiento de los existentes. A este problema lo llamamos fragmentación externa [R69]; ver Figura 16.6 (izquierda).

En el ejemplo, aparece un proceso y desea asignar un segmento de 20 KB. En ese ejemplo, hay 24 KB libres, pero no en un segmento contiguo (más bien, en tres fragmentos no contiguos). Por lo tanto, el sistema operativo no puede satisfacer la solicitud de 20 KB. Podrían ocurrir problemas similares cuando llega una solicitud para hacer crecer un segmento; Si los siguientes bytes de espacio físico no están disponibles, el sistema operativo tendrá que rechazar la solicitud, aunque pueda haber bytes libres disponibles en otras partes de la memoria física.

Una solución a este problema sería compactar la memoria física reorganizando los segmentos existentes. Por ejemplo, el sistema operativo podría detener cualquier proceso que se esté ejecutando, copiar sus datos a una región contigua de memoria, cambiar los valores de sus registros de segmento para que apunten a las nuevas ubicaciones físicas y así tener una gran cantidad de memoria libre con la que trabajar. . Al hacerlo, el sistema operativo permite que la nueva solicitud de asignación tenga éxito. Sin embargo, la compactación es costosa, ya que copiar segmentos consume mucha memoria y generalmente utiliza una buena cantidad de tiempo de procesador; ver

Figura 16.6 (derecha) para ver un diagrama de memoria física compactada. La compactación también (íronicamente) hace que las solicitudes para hacer crecer los segmentos existentes sean difíciles de satisfacer. servir, y por lo tanto puede causar una mayor reorganización para dar cabida a tales solicitudes.

En cambio, un enfoque más sencillo podría ser utilizar una gestión de listas libres. Algoritmo que intenta mantener grandes extensiones de memoria disponibles para su asignación. Hay literalmente cientos de enfoques que la gente ha adoptado, incluyendo algoritmos clásicos como best-fit (que mantiene una lista de espacios libres y devuelve el tamaño más cercano que satisface la asignación deseada a el solicitante), esquemas de peor ajuste, primero ajuste y más complejos como el de amigo algoritmo [K68]. Una excelente encuesta realizada por Wilson et al. es un buen lugar para comience si desea obtener más información sobre dichos algoritmos [W+95], o puede Espere hasta que cubramos algunos de los conceptos básicos en un capítulo posterior. Desafortunadamente, Sin embargo, no importa cuán inteligente sea el algoritmo, la fragmentación externa todavía existen; por tanto, un buen algoritmo simplemente intenta minimizarlo.

## 16.7 Resumen

La segmentación resuelve una serie de problemas y nos ayuda a construir una sociedad más virtualización efectiva de la memoria. Más allá de la simple reubicación dinámica, la segmentación puede soportar mejor espacios de direcciones dispersos, evitando los enormes potencial desperdicio de memoria entre segmentos lógicos del espacio de direcciones. También es rápido, ya que realizar la segmentación aritmética que requiere es fácil y muy adecuado para hardware; los gastos generales de traducción son mínimos. A También surge un beneficio adicional: el código compartido. Si el código se coloca dentro de un segmento separado, dicho segmento podría potencialmente compartirse entre múltiples programas en ejecución.

Sin embargo, como aprendimos, la asignación de segmentos de tamaño variable en la memoria genera algunos problemas que nos gustaría superar. La primera, como se analizó anteriormente, es la fragmentación externa. Debido a que los segmentos son de tamaño variable, la memoria libre se divide en partes de tamaño impar y, por lo tanto, satisfacer una solicitud de asignación de memoria puede resultar difícil. Se puede intentar utilizar algoritmos inteligentes [W+95] o memoria compacta periódicamente, pero el problema es fundamental y difícil de evitar.

El segundo problema, y quizás el más importante, es que la segmentación todavía no es lo suficientemente flexible para soportar nuestra dirección escasa y completamente generalizada espacio. Por ejemplo, si tenemos un montón grande pero poco utilizado, todo en uno segmento lógico, todo el montón aún debe residir en la memoria para poder ser accedido. En otras palabras, si nuestro modelo de cómo se está distribuyendo el espacio de direcciones utilizado no coincide exactamente con cómo se ha realizado la segmentación subyacente Diseñado para soportarlo, la segmentación no funciona muy bien. nosotros así Necesitamos encontrar algunas soluciones nuevas. ¿Listo para encontrarlos?

## Referencias

- [CV65] "Introducción y descripción general del sistema Multics" por FJ Corbato, VA Vyssotsky. Fall Joint Computer Conference, 1965. Uno de los cinco artículos presentados sobre Multics en la Fall Joint Computer Conference; ¡Oh, ser una mosca en la pared de esa habitación ese día!
- [DD68] "Memoria virtual, procesos y uso compartido en multics" por Robert C. Daley y Jack B. Dennis. Communications of the ACM, Volumen 11:5, mayo de 1968. Uno de los primeros artículos sobre cómo realizar enlaces dinámicos en Multics, que estaba muy adelantado a su tiempo. Los enlaces dinámicos finalmente encontraron su camino de regreso a los sistemas unos 20 años después, como lo exigían las grandes bibliotecas de X-Windows. ¡Algunos dicen que estas grandes bibliotecas X11 fueron la venganza del MIT por eliminar el soporte para enlaces dinámicos en las primeras versiones de UNIX!
- [G62] "Segmentación de hechos" por MN Greenfield. Actas del SJCC, volumen 21, mayo de 1962. Otro artículo inicial sobre segmentación; tan temprano que no tiene referencias a otros trabajos.
- [H61] "Organización de programas y mantenimiento de registros para almacenamiento dinámico" por AW Holt. Communications of the ACM, Volumen 4:10, octubre de 1961. Un artículo increíblemente temprano y difícil de leer sobre la segmentación y algunos de sus usos.
- [I09] "Manuales para desarrolladores de software de arquitecturas Intel 64 e IA-32" de Intel. 2009. Disponible: <http://www.intel.com/products/processor/manuals>. Intenta leer sobre segmentación aquí (Capítulo 3 del Volumen 3a); Te dolerá la cabeza, al menos un poquito.
- [K68] "El arte de la programación informática: Volumen I" de Donald Knuth. Addison-Wesley, 1968. Knuth es famoso no sólo por sus primeros libros sobre el arte de la programación informática, sino también por su sistema de composición tipográfica TeX, que sigue siendo una poderosa herramienta de composición tipográfica utilizada por los profesionales de hoy en día y, de hecho, para componer este mismo libro. Sus tomos sobre algoritmos son una excelente referencia temprana a muchos de los algoritmos que subyacen a los sistemas informáticos actuales.
- [L83] "Consejos para el diseño de sistemas informáticos" de Butler Lampson. ACM Operating Systems Review, 15:5, octubre de 1983. Un tesoro de sabios consejos sobre cómo construir sistemas. Difícil de leer de una sola vez; tómalo poco a poco, como un buen vino o un manual de referencia.
- [LL82] "Gestión de memoria virtual en el sistema operativo VAX/VMS" por Henry M. Levy, Peter H. Lipman. IEEE Computer, Volumen 15:3, marzo de 1982. Un sistema de gestión de memoria clásico, con mucho sentido común en su diseño. Lo estudiaremos con más detalle en un capítulo posterior.
- [RK68] "Sistemas dinámicos de asignación de almacenamiento" por B. Randell y CJ Kuehner. Communications of the ACM, Volumen 11:5, mayo de 1968. Una buena descripción general de las diferencias entre paginación y segmentación, con alguna discusión histórica de varias máquinas.
- [R69] "Una nota sobre la fragmentación del almacenamiento y la segmentación de programas" por Brian Randell. Communications of the ACM, Volumen 12:7, julio de 1969. Uno de los primeros artículos en discutir la fragmentación.
- [W+95] "Asignación dinámica de almacenamiento: estudio y revisión crítica" por Paul R. Wilson, Mark S. Johnstone, Michael Neely, David Boles. Taller internacional sobre gestión de la memoria, Escocia, Reino Unido, septiembre de 1995. Un excelente estudio sobre asignadores de memoria.

## Tarea (Simulación)

Este programa le permite ver cómo se realizan las traducciones de direcciones en un sistema con segmentación. Consulte el archivo LÉAME para obtener más detalles.

### Preguntas

1. Primero, usemos un pequeño espacio de direcciones para traducir algunas direcciones. Aquí hay un conjunto simple de parámetros con algunas semillas aleatorias diferentes; ¿Puedes traducir las direcciones?

```
segmentación.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 0 segmentación.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20  
-s 1 segmentación.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 2
```

2. Ahora, veamos si entendemos este pequeño espacio de direcciones que hemos construido (usando los parámetros de la pregunta anterior). ¿Cuál es la dirección virtual legal más alta en el segmento 0? ¿Qué pasa con la dirección virtual legal más baja en el segmento 1? ¿Cuáles son las direcciones ilegales más bajas y más altas en todo este espacio de direcciones? Finalmente, ¿cómo ejecutarías segmentation.py con el indicador -A para comprobar si tienes razón?

3. Digamos que tenemos un pequeño espacio de direcciones de 16 bytes en una memoria física de 128 bytes. ¿Qué base y límites establecería para que el simulador genere los siguientes resultados de traducción para el flujo de direcciones especificado: válido, válido, infracción, ..., infracción, válido, válido? Suponga los siguientes parámetros:

```
segmentación.py -a 16 -p 128  
-A 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 --b0 ? --l0? --b1? --l1?
```

4. Supongamos que queremos generar un problema en el que aproximadamente el 90% de las direcciones virtuales generadas aleatoriamente sean válidas (no violaciones de segmentación). ¿Cómo deberías configurar el simulador para hacerlo? ¿Qué parámetros son importantes para obtener este resultado?

5. ¿Puedes ejecutar el simulador de manera que ninguna dirección virtual sea válida? ¿Cómo?

## Gestión del espacio libre

En este capítulo, nos desviamos un poco de nuestra discusión sobre la virtualización de la memoria para discutir un aspecto fundamental de cualquier sistema de administración de memoria, ya sea una biblioteca malloc (que administra páginas del montón de un proceso) o el propio sistema operativo. (administrar partes del espacio de direcciones de un proceso). Específicamente, discutiremos las cuestiones relacionadas con el espacio libre. gestión.

Hagamos el problema más específico. Administrar el espacio libre ciertamente puede ser fácil, como veremos cuando analicemos el concepto de paginación. Es fácil cuando el espacio que gestiona está dividido en unidades de tamaño fijo; en tal caso, simplemente mantenga una lista de estas unidades de tamaño fijo; cuando un cliente solicita uno de ellos, devuelve la primera entrada.

Donde la gestión del espacio libre se vuelve más difícil (e interesante) es cuando el espacio libre que se gestiona consta de tamaños variables. unidades; esto surge en una biblioteca de asignación de memoria a nivel de usuario (como en malloc() y free()) y en un sistema operativo que gestiona la memoria física cuando se utiliza la segmentación para implementar la memoria virtual. En cualquier caso, el problema que existe se conoce como fragmentación externa: el espacio libre se corta en pequeños trozos de diferentes tamaños y, por tanto, queda fragmentado; Las solicitudes posteriores pueden fallar porque no hay ningún espacio contiguo que pueda satisfacer la solicitud, aunque la cantidad total de espacio libre exceda el tamaño de la solicitud.



La figura muestra un ejemplo de este problema. En este caso, el total el espacio libre disponible es de 20 bytes; desafortunadamente, está fragmentado en dos trozos de tamaño 10 cada uno. Como resultado, una solicitud de 15 bytes fallará incluso aunque queden 20 bytes libres. Y así llegamos al problema que se aborda en este capítulo.

## CRUX: CÓMO GESTIONAR EL ESPACIO LIBRE ¿Cómo

se debe gestionar el espacio libre al satisfacer solicitudes de tamaño variable? ¿Qué estrategias se pueden utilizar para minimizar la fragmentación? ¿Cuáles son los gastos generales de tiempo y espacio de los enfoques alternativos?

## 17.1 Supuestos

La mayor parte de esta discusión se centrará en la gran historia de los asignadores que se encuentran en las bibliotecas de asignación de memoria a nivel de usuario. Nos basamos en la excelente encuesta de Wilson [W+95], pero animamos a los lectores interesados a consultar el documento fuente para obtener más detalles<sup>1</sup>.

Suponemos una interfaz básica como la proporcionada por malloc() y free(). Específicamente, void \*malloc(size t size) toma un único parámetro, size, que es el número de bytes solicitados por la aplicación; devuelve un puntero (sin ningún tipo en particular, o un puntero vacío en la jerga C) a una región de ese tamaño (o mayor). La rutina complementaria void free(void \*ptr) toma un puntero y libera el fragmento correspondiente. Nótese la implicación de la interfaz: el usuario, al liberar el espacio, no informa a la biblioteca de su tamaño; por lo tanto, la biblioteca debe poder calcular qué tan grande es una porción de memoria cuando se le pasa solo un puntero.

Discutiremos cómo hacer esto un poco más adelante en el capítulo.

El espacio que administra esta biblioteca se conoce históricamente como montón, y la estructura de datos genérica utilizada para administrar el espacio libre en el montón es una especie de lista libre. Esta estructura contiene referencias a todos los gratuitos.

trozos de espacio en la región administrada de la memoria. Por supuesto, esta estructura de datos no tiene por qué ser una lista per se, sino simplemente algún tipo de estructura de datos para rastrear el espacio libre.

Suponemos además que lo que nos preocupa principalmente es la fragmentación externa, como se describió anteriormente. Por supuesto, los asignadores también podrían tener el problema de la fragmentación interna; Si un asignador entrega fragmentos de memoria más grandes que el solicitado, cualquier espacio no solicitado (y por lo tanto no utilizado) en dicho fragmento se considera fragmentación interna (porque el desperdicio ocurre dentro de la unidad asignada) y es otro ejemplo de desperdicio de espacio. Sin embargo, en aras de la simplicidad y porque es el más interesante de los dos tipos de fragmentación, nos centraremos principalmente en la fragmentación externa.

También asumiremos que una vez que se entrega la memoria a un cliente, no se puede reubicar en otra ubicación de la memoria. Por ejemplo, si un programa llama a malloc() y se le proporciona un puntero a algún espacio dentro del montón, esa región de memoria es esencialmente "propiedad" del programa (y la biblioteca no puede moverla) hasta que el programa la devuelva a través de un correspondiente -ing llamada gratuita(). Por lo tanto, no es posible compactar el espacio libre, lo que

---

<sup>1</sup>Tiene casi 80 páginas; ¡Así que realmente tienes que estar interesado!

Sería útil para combatir la fragmentación<sup>2</sup>. Sin embargo, la compactación podría usarse en el sistema operativo para lidiar con la fragmentación al implementar la segmentación (como se discutió en dicho capítulo sobre segmentación).

Finalmente, asumiremos que el asignador administra una región contigua de bytes. En algunos casos, un asignador podría solicitar que esa región crezca; por ejemplo, una biblioteca de asignación de memoria a nivel de usuario podría llamar al kernel para hacer crecer el montón (a través de una llamada al sistema como sbrk) cuando se agota del espacio. Sin embargo, para simplificar, asumiremos que la región es una tamaño único fijo durante toda su vida.

## 17.2 Mecanismos de bajo nivel

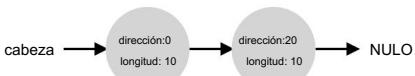
Antes de profundizar en algunos detalles de la política, primero cubriremos algunos mecanismos comunes utilizados en la mayoría de los asignadores. Primero, discutiremos los conceptos básicos de división y fusión, técnicas comunes en la mayoría de los asignadores. En segundo lugar, mostraremos cómo se puede realizar un seguimiento rápido del tamaño de las regiones asignadas. y con relativa facilidad. Finalmente, discutiremos cómo construir una lista simple dentro del espacio libre para realizar un seguimiento de lo que es gratis y lo que no.

### División y fusión

Una lista libre contiene un conjunto de elementos que describen el espacio libre que aún queda en el montón. Por tanto, supongamos el siguiente montón de 30 bytes:



La lista libre para este montón tendría dos elementos. Una entrada describe el primer segmento libre de 10 bytes (bytes 0-9) y una entrada describe el otro segmento libre (bytes 20-29):



Como se describió anteriormente, una solicitud de algo mayor que 10 bytes fallar (devolviendo NULL); simplemente no hay un solo fragmento contiguo de memoria de ese tamaño disponible. Una solicitud de exactamente ese tamaño (10 bytes) podría ser satisfecha fácilmente con cualquiera de los trozos libres. Pero ¿qué pasa si el

¿La solicitud es para algo menor que 10 bytes?

Supongamos que tenemos una solicitud de un solo byte de memoria. en esto En este caso, el asignador realizará una acción conocida como división: encontrará

<sup>2</sup>Una vez que se entrega un puntero a un fragmento de memoria a un programa en C, generalmente resulta difícil para determinar todas las referencias (punteros) a esa región, que pueden almacenarse en otras variables o incluso en registros en un momento determinado de la ejecución. Puede que este no sea el caso en lenguajes más fuertemente tipados y con recolección de basura, lo que permitiría así la compactación como técnica para fragmentación del combate.

una porción de memoria libre que puede satisfacer la solicitud y dividirla en dos. El primer fragmento lo devolverá a la persona que llama; el segundo trozo permanecerá en la lista. Por lo tanto, en nuestro ejemplo anterior, si se hiciera una solicitud de 1 byte y el asignador decidiera usar el segundo de los dos elementos de la lista para satisfacer la solicitud, la llamada a `malloc()` devolvería 20 (la dirección del Región asignada de 1 byte) y la lista terminaría luciendo así:



En la imagen, puedes ver que la lista básicamente permanece intacta; el único cambio es que la región libre ahora comienza en 21 en lugar de 20, y la longitud de esa región libre ahora es solo 93 . Por lo tanto, la división se usa comúnmente en asignadores cuando las solicitudes son más pequeñas que el tamaño de cualquier fragmento libre en particular.

Un mecanismo corolario que se encuentra en muchos asignadores se conoce como fusión de espacio libre. Tome nuestro ejemplo anterior una vez más (10 bytes libres, 10 bytes usados y otros 10 bytes libres).

Dado este (pequeño) montón, ¿qué sucede cuando una aplicación llama a `free(10)`, devolviendo así el espacio en el medio del montón? Si simplemente volvemos a agregar este espacio libre a nuestra lista sin pensar demasiado, podríamos terminar con una lista similar a esta:



Tenga en cuenta el problema: si bien todo el montón ahora está libre, aparentemente está dividido en tres fragmentos de 10 bytes cada uno. Por lo tanto, si un usuario solicita 20 bytes, un recorrido simple de la lista no encontrará ese fragmento libre y devolverá un error.

Lo que hacen los asignadores para evitar este problema es fusionar el espacio libre cuando se libera una parte de la memoria. La idea es simple: al devolver un fragmento libre en la memoria, observe cuidadosamente las direcciones del fragmento que está devolviendo, así como los fragmentos de espacio libre cercanos; Si el espacio recién liberado se encuentra justo al lado de uno (o dos, como en este ejemplo) fragmentos libres existentes, combinelos en un único fragmento libre más grande. Por lo tanto, con la fusión, nuestra lista final debería verse así:



De hecho, así es como se veía la lista del montón al principio, antes de que se realizaran asignaciones. Con la fusión, un asignador puede garantizar mejor que haya grandes extensiones libres disponibles para la aplicación.

---

<sup>3</sup>Esta discusión supone que no hay encabezados, una suposición poco realista pero simplificadora. cién que hacemos por ahora.

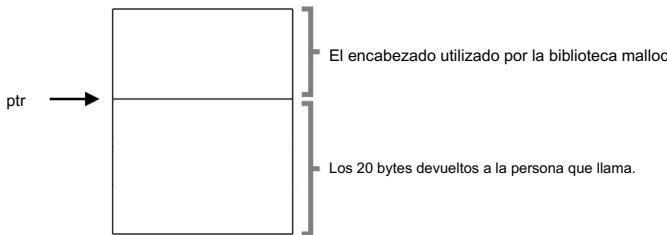


Figura 17.1: Encabezado Región Plus asignado

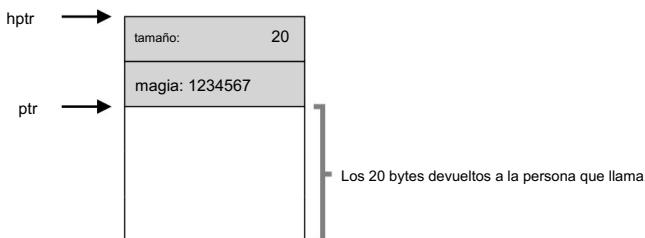


Figura 17.2: Contenidos específicos del encabezado

### Seguimiento del tamaño de las regiones asignadas

Es posible que hayas notado que la interfaz para liberar (`void *ptr`) no toma un parámetro de tamaño; por lo tanto, se supone que dado un puntero, la biblioteca malloc puede determinar rápidamente el tamaño de la región de memoria que se libera y así incorporar el espacio nuevamente a la lista libre.

Para realizar esta tarea, la mayoría de los asignadores almacenan un poco de información adicional en un bloque de encabezado que se guarda en la memoria, generalmente justo antes del fragmento de memoria entregado. Veamos un ejemplo nuevamente (Figura 17.1). En este ejemplo, estamos examinando un bloque asignado de tamaño 20 bytes, señalado por `ptr`; Imagine que el usuario llamó a `malloc()` y almacenó los resultados en `ptr`, por ejemplo, `ptr = malloc(20);`.

El encabezado contiene mínimamente el tamaño de la región asignada (en este caso, 20); También puede contener indicadores adicionales para acelerar la desasignación, un número mágico para proporcionar una verificación de integridad adicional y otra información. Supongamos un encabezado simple que contiene el tamaño de la región y un número mágico, como este:

```
estructura typedef { int
    tamaño; magia
    interna; }
encabezado_t;
```

El ejemplo anterior se parecería a lo que ve en la Figura 17.2. Cuando el usuario llama a `free(ptr)`, la biblioteca utiliza aritmética de punteros simple para determinar dónde comienza el encabezado:

```
vacio libre (vacio *ptr) {
    encabezado_t *hptr = (encabezado_t *) ptr - 1;
    ...
}
```

Después de obtener dicho puntero al encabezado, la biblioteca puede determinar fácilmente si el número mágico coincide con el valor esperado como una verificación de cordura (`assert(hptr->magic == 1234567)`) y calcular el tamaño total de la región recién liberada mediante matemáticas simples (es decir, sumando el tamaño del encabezado al tamaño de la región). Tenga en cuenta el pequeño pero fundamental detalle de la última frase: el tamaño de la región libre es el tamaño del encabezado más el tamaño del espacio asignado al usuario. Por tanto, cuando un usuario solicita N bytes de memoria, la biblioteca no busca un fragmento libre de tamaño N; más bien, busca un fragmento libre de tamaño N más el tamaño del encabezado.

#### Incorporación de una lista libre

Hasta ahora hemos tratado nuestra lista libre simple como una entidad conceptual; es sólo una lista que describe los fragmentos de memoria libres en el montón. Pero, ¿cómo construimos una lista de este tipo dentro del propio espacio libre?

En una lista más típica, al asignar un nuevo nodo, simplemente llamaría a `malloc()` cuando necesite espacio para el nodo. Desafortunadamente, dentro de la biblioteca de asignación de memoria, ¡no puedes hacer esto! En su lugar, debe crear la lista dentro del propio espacio libre. No te preocupes si esto suena un poco raro; Lo es, ¡pero no tan extraño como para que no puedas hacerlo!

Supongamos que tenemos una porción de memoria de 4096 bytes para administrar (es decir, el montón es de 4 KB). Para gestionar esto como una lista libre, primero tenemos que inicializar dicha lista; Inicialmente, la lista debe tener una entrada de tamaño 4096 (menos el tamaño del encabezado). Aquí está la descripción de un nodo de la lista:

```
typedef estructura __node_t { int tamaño;
    estructura __node_t *siguiente; } nodo_t;
```

Ahora veamos un código que inicializa el montón y coloca el primer elemento de la lista libre dentro de ese espacio. Suponemos que el montón se construye dentro de un espacio libre adquirido mediante una llamada al sistema `mmap()`; Esta no es la única forma de construir un montón de este tipo, pero nos resulta útil en este ejemplo. Aquí está el código:

```
// mmap() devuelve un puntero a una porción de espacio libre node_t *head =
mmap(NULL, 4096, PROT_READ|PROT_WRITE,
        MAP_ANON|MAP_PRIVATE, -1, 0); cabeza-
>tamaño = 4096 - tamaño de(nodo_t); encabezado->siguiente
= NULL;
```

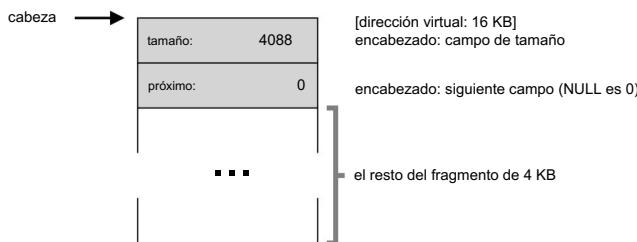


Figura 17.3: Un montón con un fragmento libre

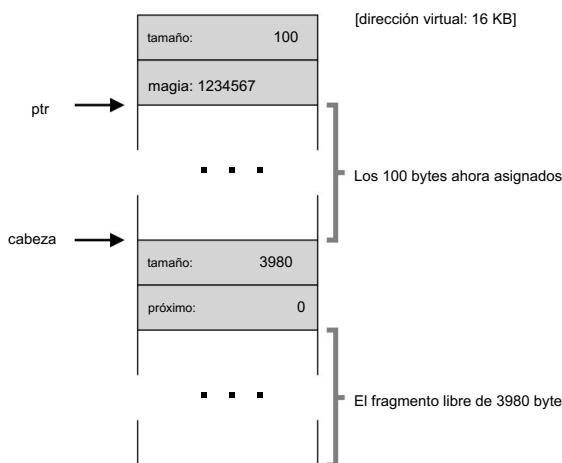


Figura 17.4: Un montón: después de una asignación

Después de ejecutar este código, el estado de la lista es que tiene una sola entrada, de tamaño 4088. Sí, este es un montón pequeño, pero nos sirve como un buen ejemplo aquí. El puntero principal contiene la dirección inicial de este rango; supongamos que son 16 KB (aunque cualquier dirección virtual estaría bien). Visualmente, el montón se parece a lo que se ve en la Figura 17.3.

Ahora, imaginemos que se solicita una porción de memoria, digamos de 100 bytes. Para atender esta solicitud, la biblioteca primero encontrará un fragmento que sea lo suficientemente grande como para acomodar la solicitud; debido a que solo hay un fragmento libre (tamaño: 4088), se elegirá este fragmento. Luego, el fragmento se dividirá en dos: un fragmento lo suficientemente grande como para atender la solicitud (y el encabezado, como se describe anteriormente) y el fragmento libre restante. Suponiendo un encabezado de 8 bytes (un tamaño entero y un número mágico entero), el espacio en el montón ahora se parece a lo que se ve en la Figura 17.4.

Por lo tanto, ante la solicitud de 100 bytes, la biblioteca asignó 108 bytes.

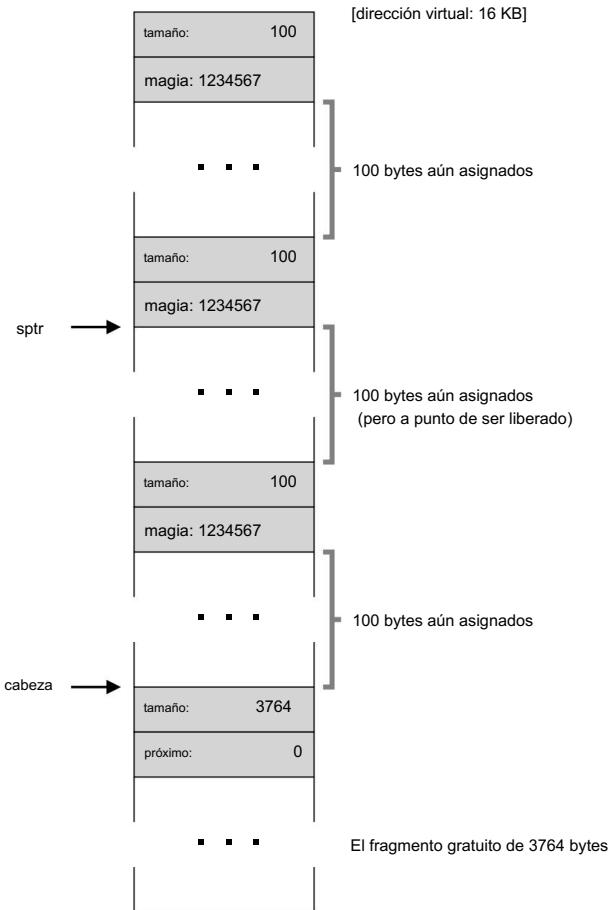


Figura 17.5: Espacio libre con tres fragmentos asignados

del fragmento libre existente, devuelve un puntero (marcado como `ptr` en el figura arriba), guarda la información del encabezado inmediatamente antes del espacio asignado para uso posterior en `free()` y reduce el nodo libre .  
en la lista a 3980 bytes (4088 menos 108).

Ahora veamos el montón cuando hay tres regiones asignadas, cada una de 100 bytes (o 108 incluyendo el encabezado). Una visualización de este montón es se muestra en la Figura 17.5.

Como puede ver allí, los primeros 324 bytes del montón ahora están asignados y, por lo tanto, vemos tres encabezados en ese espacio, así como tres regiones de 100 bytes. regiones de bytes que utiliza el programa que realiza la llamada. La lista libre permanece poco interesante: solo un nodo (señalado por la cabeza), pero ahora solo 3764

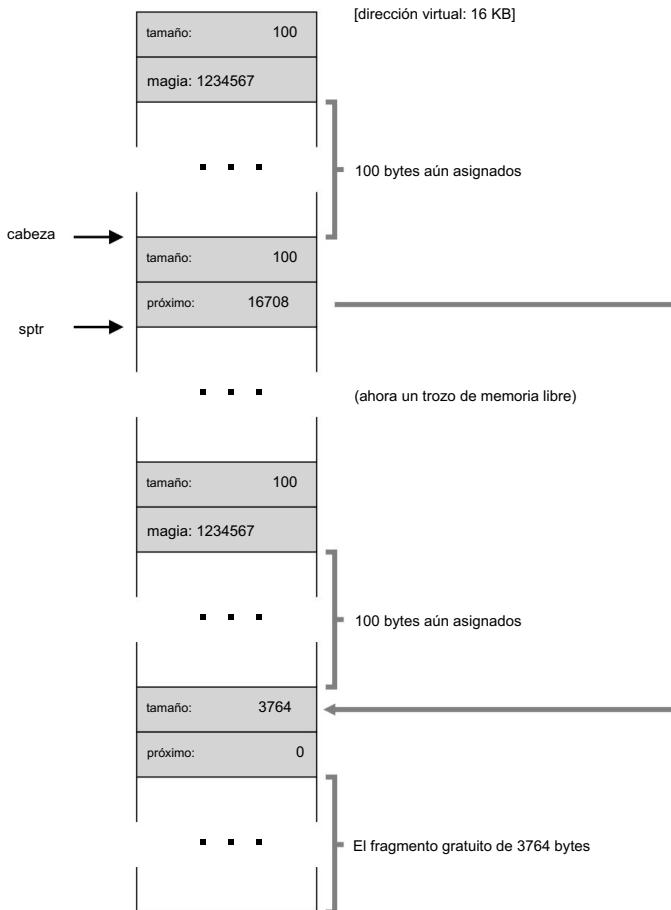


Figura 17.6: Espacio libre con dos fragmentos asignados

bytes de tamaño después de las tres divisiones. Pero, ¿qué sucede cuando el llamado ¿El programa devuelve algo de memoria a través de free()?

En este ejemplo, la aplicación devuelve la parte intermedia de la asignación memoria, llamando a free(16500) (el valor 16500 se obtiene mediante agregando el inicio de la región de memoria, 16384, a los 108 de la anterior fragmento y los 8 bytes del encabezado de este fragmento). Este valor se muestra en el diagrama anterior por el puntero sptr.

La biblioteca calcula inmediatamente el tamaño de la región libre y luego agrega el fragmento libre nuevamente a la lista libre. Suponiendo que insertamos en el encabezado de la lista libre, el espacio ahora se ve así (Figura 17.6).

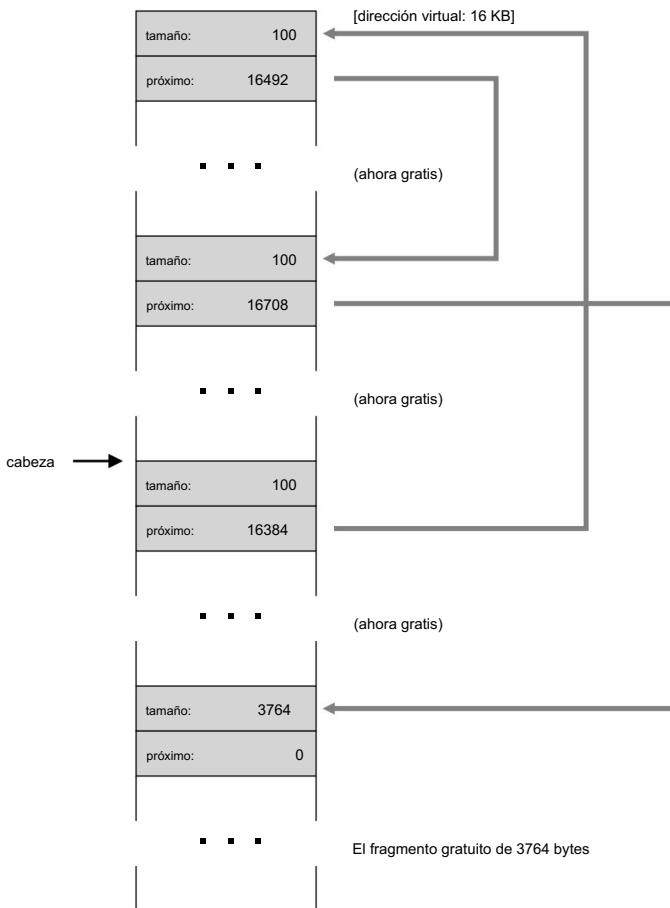


Figura 17.7: Una lista libre no fusionada

Ahora tenemos una lista que comienza con un pequeño fragmento libre (100 bytes, señalado por el encabezado de la lista) y un gran fragmento libre (3764 bytes). ¡Nuestra lista finalmente tiene más de un elemento! Y sí, el espacio libre. está fragmentado, algo desafortunado pero común.

Un último ejemplo: supongamos ahora que los dos últimos fragmentos en uso son liberados. Sin fusionarse, se termina con fragmentación (Figura 17.7).

Como puedes ver en la figura, ¡ahora tenemos un gran lío! ¿Por qué? Simple, Nos olvidamos de fusionar la lista. Aunque toda la memoria está libre, es cortado en pedazos, apareciendo así como un recuerdo fragmentado a pesar de no serlo. La solución es simple: revisa la lista y fusiona trozos vecinos; cuando termine, el montón volverá a estar completo.

### Hacer crecer el montón

Deberíamos discutir un último mecanismo que se encuentra en muchas bibliotecas de asignación. Específicamente, ¿qué debería hacer si el montón se queda sin espacio? El enfoque más simple es simplemente fallar. En algunos casos, esta es la única opción y, por lo tanto, devolver NULL es un enfoque honorable. ¡No te sientas mal! Lo intentaste y, aunque fracasaste, peleaste la buena batalla.

La mayoría de los asignadores tradicionales comienzan con un montón de tamaño pequeño y luego solicitan más memoria del sistema operativo cuando se agota. Normalmente, esto significa que hacen algún tipo de llamada al sistema (por ejemplo, sbrk en la mayoría de los sistemas UNIX) para hacer crecer el montón y luego asignan los nuevos fragmentos desde allí. Para atender la solicitud sbrk, el sistema operativo busca páginas físicas libres, las asigna al espacio de direcciones del proceso de solicitud y luego devuelve el valor del final del nuevo montón; en ese momento, hay un montón más grande disponible y la solicitud se puede atender exitosamente.

## 17.3 Estrategias básicas

Ahora que tenemos algo de maquinaria en nuestro haber, repasemos algunas estrategias básicas para gestionar el espacio libre. Estos enfoques se basan principalmente en políticas bastante simples que usted mismo podría idear; Pruébelo antes de leer y vea si se le ocurren todas las alternativas (¡o tal vez algunas nuevas!).

El asignador ideal es rápido y minimiza la fragmentación. Desafortunadamente, debido a que el flujo de asignaciones y solicitudes libres puede ser arbitrario (después de todo, están determinados por el programador), cualquier estrategia en particular puede funcionar bastante mal si se utiliza un conjunto incorrecto de entradas. Por lo tanto, no describiremos el “mejor” enfoque, sino que hablaremos de algunos conceptos básicos y discutiremos sus pros y sus contras.

### Mejor ajuste

La estrategia de mejor ajuste es bastante simple: primero, busque en la lista libre y encuentre fragmentos de memoria libre que sean tan grandes o mayores que el tamaño solicitado. Luego, devuelve el que sea más pequeño en ese grupo de candidatos; este es el llamado fragmento de mejor ajuste (también podría llamarse el ajuste más pequeño).

Una pasada por la lista libre es suficiente para encontrar el bloque correcto al que regresar.

La intuición detrás de Best Fit es simple: al devolver un bloque cercano a lo que pide el usuario, Best Fit intenta reducir el espacio desperdiciado. Sin embargo, hay un costo; Las implementaciones ingenuas pagan una gran penalización de rendimiento al realizar una búsqueda exhaustiva del bloque libre correcto.

### Peor ajuste

El enfoque de peor ajuste es lo opuesto al de mejor ajuste; encontrar la porción más grande y devolver la cantidad solicitada; mantenga la porción restante (grande) en la lista libre. El peor ajuste intenta dejar grandes trozos libres en lugar de muchos

pequeños trozos que pueden surgir de un enfoque de mejor ajuste. Una vez más, sin embargo, se requiere una búsqueda completa de espacio libre y, por lo tanto, este enfoque puede ser costoso. Peor aún, la mayoría de los estudios muestran que funciona mal, lo que lleva a un exceso fragmentación sin dejar de tener altos gastos generales.

#### Primer ajuste

El primer método de ajuste simplemente encuentra el primer bloque que sea lo suficientemente grande y devuelve el importe solicitado al usuario. Como antes, el resto libre El espacio se mantiene libre para solicitudes posteriores.

El primer ajuste tiene la ventaja de ser rápido: no es necesario realizar una búsqueda exhaustiva de todos los espacios libres son necesarios, pero a veces contamina el comienzo de la lista libre con objetos pequeños. Por lo tanto, ¿cómo gestiona el asignador la lista libre? El orden se convierte en un problema. Un enfoque es utilizar pedidos basados en direcciones; manteniendo la lista ordenada por la dirección del espacio libre, fusionándose se vuelve más fácil y la fragmentación tiende a reducirse.

#### Siguiente ajuste

En lugar de comenzar siempre la búsqueda de primer ajuste al principio de la lista, el siguiente algoritmo de ajuste mantiene un puntero adicional a la ubicación dentro del lista donde se miró por última vez. La idea es difundir las búsquedas de Liberar espacio en toda la lista de manera más uniforme, evitando así fragmentaciones. del comienzo de la lista. El rendimiento de tal enfoque es bastante similar al primer ajuste, ya que una vez más se evita una búsqueda exhaustiva.

#### Ejemplos

A continuación se muestran algunos ejemplos de las estrategias anteriores. Imagine una lista gratuita con tres elementos, de tamaños 10, 30 y 20 (ignoraremos los encabezados y otros detalles aquí, en lugar de centrarse únicamente en cómo funcionan las estrategias):



Supongamos una solicitud de asignación de tamaño 15. Un enfoque que mejor se ajuste sería busque en toda la lista y descubra que 20 era la mejor opción, ya que es la más pequeña Espacio libre que pueda acoger la petición. La lista libre resultante:



Como sucede en este ejemplo, y sucede a menudo con un enfoque de mejor ajuste, ahora queda una pequeña porción libre. Un enfoque de peor ajuste es similar pero en su lugar encuentra el fragmento más grande, en este ejemplo 30. La lista resultante:



La estrategia de primer ajuste, en este ejemplo, hace lo mismo que la de peor ajuste, y también encuentra el primer bloque libre que puede satisfacer la solicitud. La diferencia está en el costo de búsqueda; tanto el mejor ajuste como el peor ajuste revisan la lista completa; first-fit solo examina fragmentos libres hasta que encuentra uno que encaja, reduciendo así el costo de búsqueda.

Estos ejemplos apenas tocan la superficie de las políticas de asignación. Se requiere un análisis más detallado con cargas de trabajo reales y comportamientos de asignadores más complejos (por ejemplo, fusión) para una comprensión más profunda. ¿Quizás algo para una sección de tareas, dices?

## 17.4 Otros enfoques

Más allá de los enfoques básicos descritos anteriormente, se han sugerido una serie de técnicas y algoritmos para mejorar la asignación de memoria de alguna manera. Aquí enumeraremos algunos de ellos para su consideración (es decir, para que piense en algo más que la asignación que mejor se ajuste).

### Listas segregadas Un

enfoque interesante que existe desde hace algún tiempo es el uso de listas segregadas. La idea básica es simple: si una aplicación en particular tiene una (o varias) solicitudes de tamaño popular que realiza, mantenga una lista separada solo para administrar objetos de ese tamaño; todas las demás solicitudes se reenvían a un asignador de memoria más general.

Los beneficios de este enfoque son obvios. Al tener una porción de memoria dedicada a un tamaño particular de solicitudes, la fragmentación es una preocupación mucho menor; Además, las solicitudes de asignación y gratuitas se pueden atender con bastante rapidez cuando son del tamaño correcto, ya que no se requiere una búsqueda complicada en una lista.

Como cualquier buena idea, este enfoque también introduce nuevas complicaciones en el sistema. Por ejemplo, ¿cuánta memoria se debe dedicar al conjunto de memoria que atiende solicitudes especializadas de un tamaño determinado, en comparación con el conjunto general? Un asignador en particular, el asignador de losa del super ingeniero Jeff Bonwick (que fue diseñado para su uso en el kernel de Solaris), maneja este problema de una manera bastante agradable [B94].

Especificamente, cuando el kernel arranca, asigna una cantidad de cachés de objetos para los objetos del kernel que probablemente se solicitarán con frecuencia (como bloques, inodos del sistema de archivos, etc.); Por lo tanto, las cachés de objetos son listas libres segregadas de un tamaño determinado y sirven rápidamente para la asignación de memoria y las solicitudes gratuitas. Cuando una caché determinada se está quedando sin espacio libre, solicita algunos bloques de memoria de un asignador de memoria más general (la cantidad total solicitada es un múltiplo del tamaño de la página y el objeto en cuestión). Por el contrario, cuando los recuentos de referencia de los objetos dentro de una losa determinada llegan a cero, el asignador general puede reclamarlos del asignador especializado, lo que a menudo se hace cuando el sistema VM necesita más memoria.

#### ADEMÁS: LOS GRANDES INGENIEROS SON REALMENTE GRANDES

Ingenieros como Jeff Bonwick (quien no solo escribió el asignador de losas mencionado aquí sino que también fue el líder de un sorprendente sistema de archivos, ZFS) son el corazón de Silicon Valley. Detrás de casi cualquier gran producto o tecnología hay un ser humano (o un pequeño grupo de humanos) que está muy por encima del promedio en talentos, habilidades y dedicación. Como dice Mark Zuckerberg (de Facebook): "Alguien que es excepcional en su rol no es sólo un poco mejor que alguien que es bastante bueno. Son 100 veces mejores". Por eso, todavía hoy, una o dos personas pueden iniciar una empresa que cambie la faz del mundo para siempre (pensemos en Google, Apple o Facebook). Trabaja duro y podrías convertirte también en una persona "100x". En su defecto, trabaje con dicha persona; aprenderá más en un día de lo que la mayoría aprende en un mes. En su defecto, siéntete triste.

El asignador de losas también va más allá de la mayoría de los enfoques de listas segregadas al mantener los objetos libres en las listas en un estado preinicializado. Bonwick muestra que la inicialización y destrucción de estructuras de datos es costosa [B94]; Al mantener los objetos liberados en una lista particular en su estado inicializado, el asignador de losas evita ciclos frecuentes de inicialización y destrucción por objeto y, por lo tanto, reduce notablemente los gastos generales.

#### Asignación de compañeros

Debido a que la fusión es fundamental para un asignador, se han diseñado algunos enfoques para simplificar la fusión. Un buen ejemplo se encuentra en el asignador de amigos binario [K65].

En un sistema de este tipo, la memoria libre se considera primero conceptualmente como un gran espacio de tamaño 2. Cuando se realiza una solicitud de memoria, la búsqueda de espacio libre divide recursivamente el espacio libre por dos hasta que se encuentra un bloque que es lo suficientemente grande para acomodar la solicitud (y una división adicional en dos daría como resultado un espacio demasiado pequeño). En este punto, el bloque solicitado se devuelve al usuario. A continuación se muestra un ejemplo de un espacio libre de 64 KB que se divide en la búsqueda de un bloque de 7 KB (Figura 17.8, página 15).

En el ejemplo, se asigna el bloque de 8 KB más a la izquierda (como lo indica el tono de gris más oscuro) y se devuelve al usuario; tenga en cuenta que este esquema puede sufrir fragmentación interna, ya que solo se le permite entregar bloques de potencia de dos tamaños.

La belleza de la asignación de amigos se encuentra en lo que sucede cuando se libera ese bloque. Al devolver el bloque de 8 KB a la lista libre, el asignador verifica si el "compañero" de 8 KB está libre; si es así, fusiona los dos bloques en un bloque de 16 KB. Luego, el asignador verifica si el compañero del bloque de 16 KB todavía está libre; si es así, fusiona esos dos bloques. Este proceso recursivo de coalescencia continúa en el árbol, ya sea restaurando todo el espacio libre o deteniéndose cuando se descubre que un compañero está en uso.

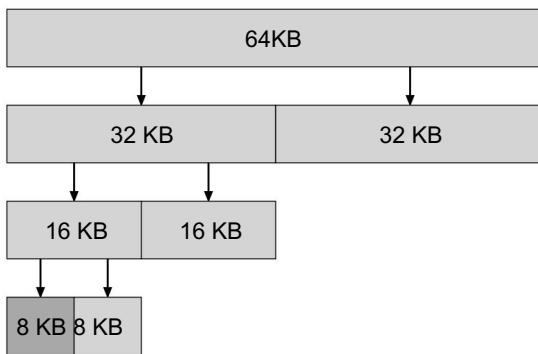


Figura 17.8: Ejemplo de montón administrado por amigos

La razón por la que la asignación de amigos funciona tan bien es que es sencillo determinar el amigo de un bloque en particular. ¿Cómo, preguntas? Piensa en las direcciones de los bloques en el espacio libre de arriba. Si piensas cuidadosamente suficiente, verás que la dirección de cada par de amigos sólo difiere por un solo bit; qué bit está determinado por el nivel en el árbol de amigos. Y De este modo, tendrá una idea básica de cómo funcionan los esquemas binarios de asignación de amigos. Para obtener más detalles, como siempre, consulte la encuesta de Wilson [W+95].

## Otras ideas

Un problema importante con muchos de los enfoques descritos anteriormente es su falta de escalamiento. Específicamente, la búsqueda de listas puede ser bastante lenta. De este modo, Los asignadores avanzados utilizan estructuras de datos más complejas para abordar estos problemas. costos, cambiando simplicidad por rendimiento. Los ejemplos incluyen árboles binarios equilibrados, árboles dispersos o árboles parcialmente ordenados [W+95].

Dado que los sistemas modernos suelen tener múltiples procesadores y ejecutar cargas de trabajo de subprocesos múltiples (algo que aprenderá con gran detalle) en la sección del libro sobre Concurrency), no es sorprendente que muchos Se ha invertido mucho esfuerzo en hacer que los asignadores funcionen bien en sistemas basados en multiprocesadores. Dos maravillosos ejemplos se encuentran en Berger et al. [B+00] y Evans [E06]; Échales un vistazo para conocer los detalles.

Estas son sólo dos de las miles de ideas que la gente ha tenido a lo largo del tiempo. sobre asignadores de memoria; lee por tu cuenta si tienes curiosidad. Defecto eso, lee sobre cómo funciona el asignador glibc [S15], para darle una idea de cómo es el mundo real.

## 17.5 Resumen

En este capítulo, hemos analizado las formas más rudimentarias de asignadores de memoria. Estos asignadores existen en todas partes, vinculados a cada programa C que usted escribe, así como en el sistema operativo subyacente que administra la memoria para sus propias estructuras de datos. Como ocurre con muchos sistemas, hay muchos

Es necesario hacer concesiones al construir un sistema de este tipo, y cuanto más sepa acerca de la carga de trabajo exacta presentada a un asignador, más podrá hacer para ajustarlo para que funcione mejor para esa carga de trabajo. Crear un asignador escalable, rápido y eficiente en el espacio que funcione bien para una amplia gama de cargas de trabajo sigue siendo un desafío constante en los sistemas informáticos modernos.

## Referencias

- [B+00] "Hoard: un asignador de memoria escalable para aplicaciones multiproceso" por Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, Paul R. Wilson. ASPLOS-IX, noviembre de 2000. Excelente asignador de Berger y compañía para sistemas multiprocesador. Más allá de ser un documento divertido, ¡también se utiliza en la práctica!
- [B94] "El asignador de losa: un asignador de memoria del kernel de almacenamiento en caché de objetos" por Jeff Bonwick. USENIX '94. Un artículo interesante sobre cómo crear un asignador para el kernel de un sistema operativo y un excelente ejemplo de cómo especializarse para tamaños de objetos comunes particulares.
- [E06] "Una implementación escalable y concurrente de malloc(3) para FreeBSD" por Jason Evans. Abril de 2006. <http://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf>. Una mirada detallada a cómo construir un asignador moderno real para usar en multiprocesadores. El asignador "jemalloc" se utiliza ampliamente en la actualidad, en FreeBSD, NetBSD, Mozilla Firefox y Facebook.
- [K65] "Un asignador de almacenamiento rápido" por Kenneth C. Knuton. Comunicaciones de la ACM, Volumen 8:10, octubre de 1965. La referencia común para la asignación de compañeros. Un hecho extraño y aleatorio: Knuth le da crédito por la idea no a Knuton sino a Harry Markowitz, un economista ganador del premio Nobel.
- Otro dato curioso: Knuth comunica todos sus correos electrónicos a través de una secretaria; él mismo no envía correos electrónicos, sino que le dice a su secretaria qué correo electrónico enviar y luego la secretaria hace el trabajo de enviarlos. Último dato de Knuth: creó TeX, la herramienta utilizada para comprender este libro. Es una pieza de software increíble<sup>4</sup>.
- [T15] "Comprensión de glibc malloc" por Sploitfun. Febrero de 2015. [sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/](http://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/). Una inmersión profunda en cómo funciona glibc malloc. Incrediblemente detallado y una lectura muy interesante.
- [W+95] "Asignación dinámica de almacenamiento: estudio y revisión crítica" por Paul R. Wilson, Mark S. Johnstone, Michael Neely, David Boles. Taller internacional sobre gestión de la memoria, Escocia, Reino Unido, septiembre de 1995. Un estudio excelente y de gran alcance sobre muchas facetas de la asignación de la memoria. ¡Demasiados detalles para entrar en este pequeño capítulo!

---

<sup>4</sup>En realidad utilizamos LaTeX, que se basa en las adiciones de Lamport a TeX, pero bastante parecido.

## Tarea (Simulación)

El programa, malloc.py, le permite explorar el comportamiento de un asignador de espacio libre simple como se describe en el capítulo. Consulte el archivo README para obtener detalles sobre su funcionamiento básico.

Preguntas 1.

- Primero ejecute las banderas -n 10 -H 0 -p BEST -s 0 para generar algunas asignaciones y asignaciones aleatorias. ¿Puedes predecir qué devolverá al-loc() / free()? ¿Puedes adivinar el estado de la lista gratuita después de cada solicitud? ¿Qué notas sobre la lista gratuita con el tiempo?
2. ¿En qué se diferencian los resultados cuando se utiliza una política de PEOR ajuste para buscar en la lista gratuita (-p PEOR)? ¿Qué cambia?
  3. ¿Qué pasa cuando se utiliza el PRIMER ajuste (-p PRIMERO)? que acelera cuando usas el primer ajuste?
  4. Para las preguntas anteriores, la forma en que se mantiene ordenada la lista puede afectar el tiempo que lleva encontrar una ubicación libre para algunas de las pólizas. Utilice los diferentes ordenamientos de listas libres (-ADDRSORT, -I TAMAÑOSOR+, -I TAMAÑOS-) para ver cómo interactúan las políticas y los ordenamientos de listas.
  5. La fusión de una lista libre puede ser muy importante. Aumente el número de asignaciones aleatorias (por ejemplo, hasta -n 1000). ¿Qué sucede con las solicitudes de asignación más grandes con el tiempo? Ejecute con y sin coalescencia (es decir, sin y con la bandera -C). ¿Qué diferencias en el resultado ves? ¿Qué tamaño tiene la lista libre a lo largo del tiempo en cada caso? ¿Importa el orden de la lista en este caso?
  6. ¿Qué sucede cuando cambias la fracción porcentual asignada -P a superior a 50? ¿Qué sucede con las asignaciones cuando se acerca a 100?  
¿Qué pasa cuando el porcentaje se acerca a 0?
  7. ¿Qué tipo de solicitudes específicas puedes realizar para generar un espacio libre muy fragmentado? Utilice el indicador -A para crear listas gratuitas fragmentadas y vea cómo las diferentes políticas y opciones cambian la organización de la lista gratuita.

## Paginación: Introducción

A veces se dice que el sistema operativo adopta uno de dos enfoques para resolver la mayoría de los problemas de gestión del espacio. El primer enfoque consiste en dividir las cosas en trozos de tamaño variable, como vimos con la segmentación en la memoria virtual.

Desafortunadamente, esta solución tiene dificultades inherentes. En particular, al dividir un espacio en porciones de diferentes tamaños, el espacio en sí puede fragmentarse y, por lo tanto, la asignación se vuelve más difícil con el tiempo.

Por tanto, puede que valga la pena considerar el segundo enfoque: dividir el espacio en partes de tamaño fijo. En la memoria virtual, llamamos a esta idea paginación y se remonta a un sistema temprano e importante, el Atlas [KE+62, L78].

En lugar de dividir el espacio de direcciones de un proceso en un número determinado de segmentos lógicos de tamaño variable (por ejemplo, código, montón, pila), lo dividimos en unidades de tamaño fijo, a cada una de las cuales llamamos página . En consecuencia, consideraremos la memoria física como un conjunto de ranuras de tamaño fijo llamadas marcos de página; cada uno de estos marcos puede contener una única página de memoria virtual. Nuestro desafío:

### EL CRUCERO: CÓMO VIRTUALIZAR LA MEMORIA CON PÁGINAS

¿Cómo podemos virtualizar la memoria con páginas para evitar los problemas de segmentación? ¿Cuáles son las técnicas básicas? ¿Cómo podemos hacer que esas técnicas funcionen bien, con mínimos gastos de espacio y tiempo?

#### 18.1 Un ejemplo sencillo y una descripción general

Para ayudar a que este enfoque sea más claro, ilustrémoslo con un ejemplo sencillo. La Figura 18.1 (página 2) presenta un ejemplo de un pequeño espacio de direcciones, de sólo 64 bytes de tamaño total, con cuatro páginas de 16 bytes (páginas virtuales 0, 1, 2 y 3). Los espacios de direcciones reales son mucho más grandes, por supuesto, comúnmente 32 bits y, por lo tanto, 4 GB de espacio de direcciones, o incluso 64 bits<sup>1</sup> ; En el libro, a menudo usaremos pequeños ejemplos para que sean más fáciles de digerir.

<sup>1</sup>Es difícil imaginar un espacio de direcciones de 64 bits por su tamaño sorprendente. Una analogía podría ayudar: si piensa que un espacio de direcciones de 32 bits tiene el tamaño de una cancha de tenis, un espacio de direcciones de 64 bits tiene aproximadamente el tamaño de Europa (!).

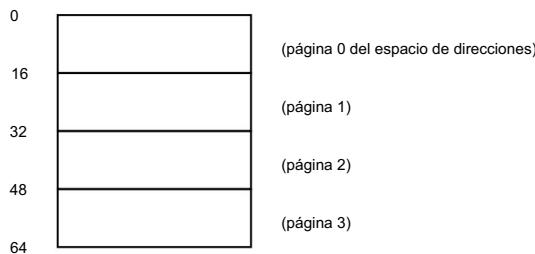


Figura 18.1: Un espacio de direcciones simple de 64 bytes

La memoria física, como se muestra en la Figura 18.2, también consta de una serie de ranuras de tamaño fijo, en este caso ocho marcos de página (lo que da como resultado una memoria física de 128 bytes, también ridículamente pequeña). Como puede ver en el diagrama, las páginas del espacio de direcciones virtuales se han colocado en diferentes ubicaciones a lo largo de la memoria física; El diagrama también muestra el sistema operativo usando parte de la memoria física para sí mismo.

La paginación, como veremos, tiene una serie de ventajas sobre nuestros enfoques anteriores. Probablemente la mejora más importante será la flexibilidad: con un enfoque de paginación completamente desarrollado, el sistema podrá soportar la abstracción de un espacio de direcciones de manera efectiva, independientemente de cómo un proceso utilice el espacio de direcciones; Por ejemplo, no haremos suposiciones sobre la dirección en la que crecen el montón y la pila y cómo se utilizan.

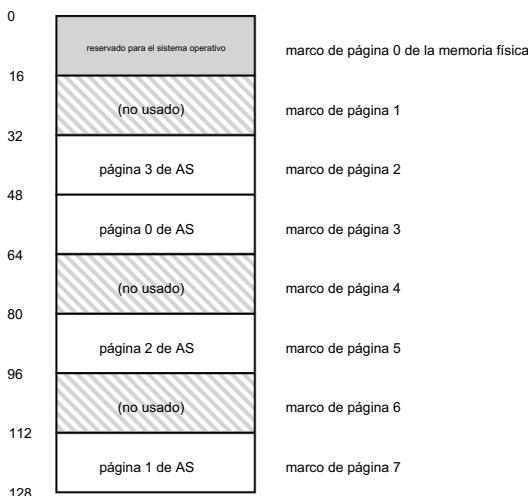


Figura 18.2: Un espacio de direcciones de 64 bytes en una memoria física de 128 bytes

Otra ventaja es la simplicidad de la gestión del espacio libre que ofrece la paginación. Por ejemplo, cuando el sistema operativo desea colocar nuestro pequeño espacio de direcciones de 64 bytes en nuestra memoria física de ocho páginas, simplemente encuentra cuatro páginas libres; tal vez el sistema operativo mantenga una lista gratuita de todas las páginas gratuitas para esto y simplemente elimine las primeras cuatro páginas gratuitas de esta lista. En el ejemplo, el sistema operativo ha colocado la página virtual 0 del espacio de direcciones (AS) en el marco físico 3, la página virtual 1 del AS en el marco físico 7, la página 2 en el marco 5 y la página 3 en el marco 2. Marcos de página 1, 4 y 6 son actualmente gratuitos.

Para registrar dónde se coloca cada página virtual del espacio de direcciones en la memoria física, el sistema operativo generalmente mantiene una estructura de datos por proceso conocida como tabla de páginas. La función principal de la tabla de páginas es almacenar traducciones de direcciones para cada una de las páginas virtuales del espacio de direcciones, permitiéndonos saber en qué parte de la memoria física reside cada página.

Para nuestro ejemplo simple (Figura 18.2, página 2), la tabla de páginas tendría las siguientes cuatro entradas: (Página virtual 0 → Marco físico 3), (VP 1 → PF 7), (VP 2 → PF 5) y (VP 3 → PF 2).

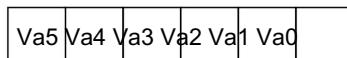
Es importante recordar que esta tabla de páginas es una estructura de datos por proceso (la mayoría de las estructuras de tablas de páginas que analizamos son estructuras por proceso; una excepción que mencionaremos es la tabla de páginas invertida). Si se ejecutara otro proceso en nuestro ejemplo anterior, el sistema operativo tendría que administrar una tabla de páginas diferente, ya que sus páginas virtuales obviamente se asignan a diferentes páginas físicas (módulo, cualquier intercambio).

Ahora sabemos lo suficiente para realizar un ejemplo de traducción de direcciones. Imaginemos que el proceso con ese pequeño espacio de direcciones (64 bytes) está realizando un acceso a la memoria:

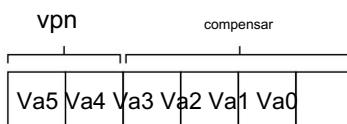
```
movl <dirección virtual>, %eax
```

Especificamente, prestemos atención a la carga explícita de los datos de la dirección <dirección virtual> en el registro eax (y así ignoremos la búsqueda de instrucciones que debe haber ocurrido antes).

Para traducir esta dirección virtual que generó el proceso, primero debemos dividirla en dos componentes: el número de página virtual (VPN) y el desplazamiento dentro de la página. Para este ejemplo, debido a que el espacio de direcciones virtuales del proceso es de 64 bytes, necesitamos 6 bits en total para nuestra dirección virtual ( $2^6 = 64$ ). Así, nuestra dirección virtual <sup>6</sup> se puede conceptualizar de la siguiente manera:



En este diagrama, Va5 es el bit de orden más alto de la dirección virtual y Va0 el bit de orden más bajo. Como conocemos el tamaño de la página (16 bytes), podemos dividir aún más la dirección virtual de la siguiente manera:

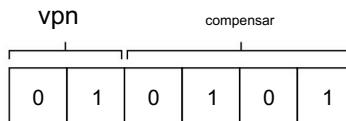


El tamaño de la página es de 16 bytes en un espacio de direcciones de 64 bytes; por lo tanto necesitamos poder seleccionar 4 páginas, y los 2 bits superiores de la dirección hacen precisamente eso. Así, tenemos un número de página virtual (VPN) de 2 bits. Los bits restantes dicen nos indica qué byte de la página nos interesa, 4 bits en este caso; llamamos esta es la compensación.

Cuando un proceso genera una dirección virtual, el sistema operativo y el hardware deben combinarse para traducirlo en una dirección física significativa. Por ejemplo, supongamos que la carga anterior fue a la dirección virtual 21:

movimiento 21, %eax

Al convertir "21" en formato binario, obtenemos "010101" y, por lo tanto, podemos examinar esta dirección virtual y ver cómo se descompone en una página virtual, número (VPN) y compensación:



Por lo tanto, la dirección virtual "21" está en el quinto byte ("0101") de la dirección virtual página "01" (o 1). Con nuestro número de página virtual, ahora podemos indexar nuestra tabla de páginas y encuentre en qué marco físico reside la página virtual 1. En la tabla de páginas encima del número de fotograma físico (PFN) (a veces también llamado número de página física o PPN) es 7 (binario 111). Así, podemos traduzca esta dirección virtual reemplazando la VPN con el PFN y luego emita la carga a la memoria física (Figura 18.3).

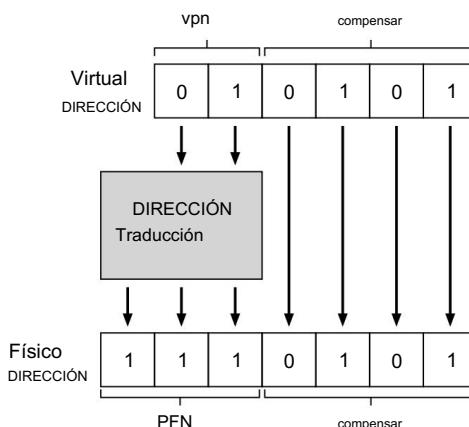


Figura 18.3: El proceso de traducción de direcciones

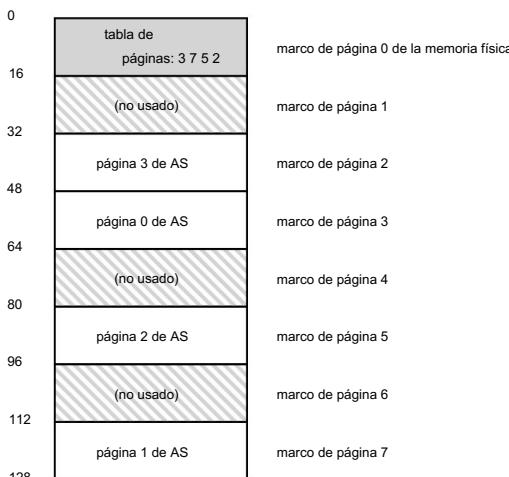


Figura 18.4: Ejemplo: Tabla de páginas en la memoria física del kernel

Tenga en cuenta que el desplazamiento permanece igual (es decir, no se traduce), porque el desplazamiento simplemente nos dice qué byte dentro de la página queremos. Nuestra dirección física final es 1110101 (117 en decimal) y es exactamente de donde queremos que nuestra carga obtenga datos (Figura 18.2, página 2).

Con esta descripción general básica en mente, ahora podemos hacer (y, con suerte, responder) algunas preguntas básicas que pueda tener sobre la paginación. Por ejemplo, ¿dónde se almacenan estas tablas de páginas? ¿Cuáles son los contenidos típicos de la tabla de páginas y qué tamaño tienen las tablas? ¿La paginación hace que el sistema sea (demasiado) lento? Estas y otras preguntas seductoras se responden, al menos en parte, en el texto siguiente. ¡Sigue leyendo!

## 18.2 ¿Dónde se almacenan las tablas de páginas?

Las tablas de páginas pueden volverse terriblemente grandes, mucho más grandes que la pequeña tabla de segmentos o el par base/límites que hemos discutido anteriormente. Por ejemplo, imagine un espacio de direcciones típico de 32 bits, con páginas de 4 KB. Esta dirección virtual se divide en una VPN de 20 bits y un desplazamiento de 12 bits (recuerde que se necesitarían 10 bits para un tamaño de página de 1 KB, y simplemente agregue dos más para llegar a 4 KB).

Una VPN de 20 bits implica que hay 2<sup>20</sup> traducciones que el sistema operativo tendría que gestionar para cada proceso (eso es aproximadamente un millón); Suponiendo que necesitamos 4 bytes por entrada de la tabla de páginas (PTE) para contener la traducción física más cualquier otro material útil, obtenemos una inmensa memoria necesaria de 4 MB para cada tabla de páginas. Eso es bastante grande. Ahora imagine que hay 100 procesos en ejecución: ¡esto significa que el sistema operativo necesitaría 400 MB de memoria solo para todas esas traducciones de direcciones! Incluso en la era moderna, donde

## APARTE: ESTRUCTURA DE DATOS : LA TABLA DE PÁGINAS Una

de las estructuras de datos más importantes en el subsistema de administración de memoria de un sistema operativo moderno es la tabla de páginas. En general, una tabla de páginas almacena traducciones de direcciones virtuales a físicas, lo que le permite al sistema saber dónde reside realmente cada página de un espacio de direcciones en la memoria física. Debido a que cada espacio de direcciones requiere dichas traducciones, en general hay una tabla de páginas por proceso en el sistema. La estructura exacta de la tabla de páginas está determinada por el hardware (sistemas más antiguos) o puede ser administrada de manera más flexible por el sistema operativo (sistemas modernos).

Las máquinas tienen gigabytes de memoria, parece un poco loco usar una gran parte solo para traducciones, ¿no? Y ni siquiera pensaremos en el tamaño que tendría dicha tabla de páginas para un espacio de direcciones de 64 bits; eso sería demasiado espantoso y quizás te asustaría por completo.

Debido a que las tablas de páginas son tan grandes, no guardamos ningún hardware especial en el chip en la MMU para almacenar la tabla de páginas del proceso que se está ejecutando actualmente. En cambio, almacenamos la tabla de páginas para cada proceso en algún lugar de la memoria. Supongamos por ahora que las tablas de páginas se encuentran en la memoria física que administra el sistema operativo; Más adelante veremos que gran parte de la memoria del sistema operativo se puede virtualizar y, por lo tanto, las tablas de páginas se pueden almacenar en la memoria virtual del sistema operativo (e incluso intercambiarse en el disco), pero eso es demasiado confuso en este momento, así que lo veremos. No lo hagas. En la Figura 18.4 (página 5) hay una imagen de una tabla de páginas en la memoria del sistema operativo; ¿Ves el pequeño conjunto de traducciones que hay ahí?

### 18.3 ¿Qué hay realmente en la tabla de páginas?

Hablemos un poco sobre la organización de la tabla de páginas. La tabla de páginas es solo una estructura de datos que se utiliza para asignar direcciones virtuales (o, en realidad, números de páginas virtuales) a direcciones físicas (números de marco físico). Por tanto, cualquier estructura de datos podría funcionar. La forma más simple se llama tabla de páginas lineal, que es simplemente una matriz. El sistema operativo indexa la matriz por el número de página virtual (VPN) y busca la entrada de la tabla de páginas (PTE) en ese índice para encontrar el número de marco físico (PFN) deseado. Por ahora, asumiremos esta estructura lineal simple; En capítulos posteriores, utilizaremos estructuras de datos más avanzadas para ayudar a resolver algunos problemas con la paginación.

En cuanto al contenido de cada PTE, tenemos varios bits diferentes que vale la pena comprender en algún nivel. Un bit válido es común para indicar si una traducción particular es válida; por ejemplo, cuando un programa comienza a ejecutarse, tendrá el código y el montón en un extremo de su espacio de direcciones y la pila en el otro. Todo el espacio no utilizado en el medio se marcará como no válido y, si el proceso intenta acceder a dicha memoria, generará una trampa para el sistema operativo que probablemente terminará el proceso.

Por tanto, el bit válido es crucial para soportar un espacio de direcciones escaso; Simplemente marcando como no válidas todas las páginas no utilizadas en el espacio de direcciones, eliminamos la necesidad de asignar marcos físicos para esas páginas y así ahorraremos una gran cantidad de memoria.

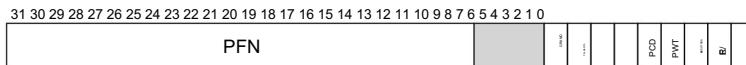


Figura 18.5: Entrada de tabla de páginas (PTE) x86

También podemos tener bits de protección, que indican si la página se puede leer, escribir o ejecutar. Nuevamente, acceder a una página de una manera no permitida por estos bits generará una trampa para el sistema operativo.

Hay un par de partes más que son importantes, pero no hablaremos mucho de ellas por ahora. Un bit presente indica si esta página está en la memoria física o en el disco (es decir, se ha intercambiado). Comprenderemos mejor esta maquinaria cuando estudiemos cómo intercambiar partes del espacio de direcciones al disco para soportar espacios de direcciones que son más grandes que la memoria física; El intercambio permite que el sistema operativo libere memoria física moviendo páginas poco utilizadas al disco. También es común un bit sucio , que indica si la página se ha modificado desde que se guardó en la memoria.

A veces se utiliza un bit de referencia (también conocido como bit de acceso) para rastrear si se ha accedido a una página y es útil para determinar qué páginas son populares y, por lo tanto, deben mantenerse en la memoria; dicho conocimiento es fundamental durante el reemplazo de páginas, un tema que estudiaremos con gran detalle en los capítulos siguientes.

La Figura 18.5 muestra una entrada de tabla de páginas de ejemplo de la arquitectura x86 [I09]. Contiene un bit presente (P); un bit de lectura/escritura (R/W) que determina si se permiten escrituras en esta página; un bit de usuario/supervisor (U/S) que determina si los procesos en modo de usuario pueden acceder a la página; algunos bits (PWT, PCD, PAT y G) que determinan cómo funciona el almacenamiento en caché de hardware para estas páginas; un bit accedido (A) y un bit sucio (D); y finalmente, el propio número de marco de página (PFN).

Lea los manuales de arquitectura Intel [I09] para obtener más detalles sobre la compatibilidad con paginación x86. Tenga cuidado, sin embargo; Leer manuales como estos, si bien es bastante informativo (y ciertamente necesario para quienes escriben código para utilizar dichas tablas de páginas en el sistema operativo), puede resultar un desafío al principio. Se requiere un poco de paciencia y muchas ganas.

#### APARTE: ¿POR QUÉ NO HAY UN BIT VÁLIDO ?

Puede notar que en el ejemplo de Intel, no hay bits válidos y presentes separados, sino solo un bit presente (P). Si ese bit está establecido ( $P=1$ ), significa que la página está presente y es válida. Si no ( $P=0$ ), significa que la página puede no estar presente en la memoria (pero es válida) o puede no ser válida. Un acceso a una página con  $P=0$  activará una trampa en el sistema operativo; Luego, el sistema operativo debe usar estructuras adicionales que mantiene para determinar si la página es válida (y por lo tanto quizás debería volver a intercambiarse) o no (y por lo tanto el programa está intentando acceder a la memoria ilegalmente). Este tipo de prudencia es común en el hardware, que a menudo sólo proporciona el conjunto mínimo de características sobre las cuales el sistema operativo puede construir un servicio completo.

## 18.4 Paginación: también demasiado lenta

Con las tablas de páginas en la memoria, ya sabemos que pueden ser demasiado grandes. Resulta que también pueden ralentizar las cosas. Por ejemplo, tome nuestras sencillas instrucciones:

```
movimiento 21, %eax
```

Nuevamente, simplemente examinemos la referencia explícita a la dirección 21 y no nos preocupemos por la búsqueda de instrucciones. En este ejemplo, asumiremos que el hardware realiza la traducción por nosotros. Para recuperar los datos deseados, el sistema primero debe traducir la dirección virtual (21) a la dirección física correcta (117). Por lo tanto, antes de recuperar los datos de la dirección 117, el sistema primero debe recuperar la entrada de la tabla de páginas adecuada de la tabla de páginas del proceso, realizar la traducción y luego cargar los datos desde la memoria física.

Para hacerlo, el hardware debe saber dónde está la tabla de páginas del proceso que se está ejecutando actualmente. Supongamos por ahora que un único registro base de tabla de páginas contiene la dirección física de la ubicación inicial de la tabla de páginas. Para encontrar la ubicación de la PTE deseada, el hardware realizará las siguientes funciones:

```
vpn           = (Dirección virtual y VPN_MASK) >> MAYÚS
PTEAddr = PageTableBaseRegister + (VPN * tamaño de (PTE))
```

En nuestro ejemplo, VPN MASK se establecería en 0x30 (hexadecimal 30 o binario 110000), lo que selecciona los bits de VPN de la dirección virtual completa; SHIFT se establece en 4 (el número de bits en el desplazamiento), de modo que movemos los bits de VPN hacia abajo para formar el número de página virtual entero correcto. Por ejemplo, con la dirección virtual 21 (010101) y el emmascaramiento convierte este valor en 010000; el turno la convierte en 01, o página virtual 1, según se deseé. Luego usamos este valor como índice en la matriz de PTE a las que apunta el registro base de la tabla de páginas.

Una vez que se conoce esta dirección física, el hardware puede recuperar el PTE de la memoria, extraer el PFN y concatenarlo con el desplazamiento de la dirección virtual para formar la dirección física deseada. Específicamente, puede pensar en que el PFN se desplaza hacia la izquierda con SHIFT y luego se realiza una operación OR bit a bit con el desplazamiento para formar la dirección final de la siguiente manera:

```
desplazamiento = Dirección virtual y OFFSET_MASK
PhysAddr = (PFN << MAYÚS) | compensar
```

Finalmente, el hardware puede recuperar los datos deseados de la memoria y colocarlos en el registro eax. ¡El programa ahora ha logrado cargar un valor de la memoria!

Para resumir, ahora describimos el protocolo inicial de lo que sucede en cada referencia de memoria. La figura 18.6 (página 9) muestra el enfoque. Para cada referencia de memoria (ya sea una recuperación de instrucciones o una carga o almacenamiento explícito), la paginación requiere que realicemos una referencia de memoria adicional para poder recuperar primero la traducción de la tabla de páginas. eso es mucho

```

1 // Extrae la VPN de la dirección virtual
2 VPN = (Dirección virtual y VPN_MASK) >> CAMBIO
3
4 // Forma la dirección de la entrada de la tabla de páginas (PTE)
5 PTEAddr = PTBR + (VPN * tamaño de (PTE))
6
7 // Busca el PTE
8 PTE = Memoria de acceso (PTEAddr)
9
10 // Comprobar si el proceso puede acceder a la página 11 if (PTE.Valid
== False)
11             RaiseException(SEGMENTATION_FAULT) 13 más si
(CanAccess(PTE.ProtectBits) == False)
12             RaiseException(PROTECTION_FAULT)
13 más
14             // El acceso está bien: forme la dirección física y recuperela offset = VirtualAddress &
15             OFFSET_MASK PhysAddr = (PTE.PFN << PFN_SHIFT) | Registro
16             de compensación = AccessMemory (PhysAddr)
17
18
19

```

Figura 18.6: Acceso a la memoria con paginación

¡trabajar! Las referencias de memoria adicionales son costosas y, en este caso, probablemente ralentizarán el proceso en un factor de dos o más.

Y ahora, con suerte, podrán ver que hay dos problemas reales que debemos resolver. Sin un diseño cuidadoso tanto del hardware como del software, las tablas de páginas harán que el sistema funcione demasiado lento, además de consumir demasiada memoria. Si bien aparentemente es una gran solución para nuestras necesidades de virtualización de memoria, primero se deben superar estos dos problemas cruciales.

## 18.5 Un rastro de memoria

Antes de cerrar, ahora rastrearemos un ejemplo simple de acceso a la memoria para demostrar todos los accesos a la memoria resultantes que ocurren cuando se usa la paginación. El fragmento de código (en C, en un archivo llamado array.c) que nos interesa es el siguiente:

```

matriz int[1000];
...
para (i = 0; i < 1000; i++) matriz[i] = 0;

```

Compilamos array.c y lo ejecutamos con los siguientes comandos:

```
indicador> gcc -o matriz array.c -Wall -O indicador> ./array
```

Por supuesto, para comprender realmente qué accesos a la memoria generará este fragmento de código (que simplemente inicializa una matriz), tendremos que saber (o asumir) algunas cosas más. Primero, tendremos que desmontar el binario resultante (usando objdump en Linux u otool en Mac) para ver qué instrucciones de ensamblaje se usan para inicializar la matriz en un bucle. Aquí está el código ensamblador resultante:

```
1024 movl $0x0,(%edi,%eax,4) 1028 incl %eax 1032 cmpl
$0x03e8,%eax 1036 jne
0x1024
```

El código, si conoce un poco de x86, es bastante fácil de entender<sup>2</sup>. La primera instrucción mueve el valor cero (que se muestra como \$0x0) a la dirección de memoria virtual de la ubicación de la matriz; esta dirección se calcula tomando el contenido de %edi y sumándole %eax multiplicado por cuatro. Por lo tanto, %edi contiene la dirección base de la matriz, mientras que %eax contiene el índice de la matriz (i); multiplicamos por cuatro porque la matriz es una matriz de números enteros, cada uno de un tamaño de cuatro bytes.

La segunda instrucción incrementa el índice de matriz mantenido en %eax, y la tercera instrucción compara el contenido de ese registro con el valor hexadecimal 0x03e8, o decimal 1000. Si la comparación muestra que dos valores aún no son iguales (que es lo que jne prueba de instrucción), la cuarta instrucción vuelve a la parte superior del bucle.

Para comprender qué memoria accede a esta secuencia de instrucciones (tanto en el nivel virtual como en el físico), tendremos que asumir algo sobre en qué parte de la memoria virtual se encuentran el fragmento de código y la matriz, así como el contenido y la ubicación de la tabla de páginas.

Para este ejemplo, asumimos un espacio de direcciones virtuales de 64 KB de tamaño (irrealmente pequeño). También asumimos un tamaño de página de 1 KB.

Todo lo que necesitamos saber ahora es el contenido de la tabla de páginas y su ubicación en la memoria física. Supongamos que tenemos una tabla de páginas lineal (basada en matrices) y que está ubicada en la dirección física 1 KB (1024).

En cuanto a su contenido, solo hay unas pocas páginas virtuales de las que debemos preocuparnos y mapearlas para este ejemplo. Primero, está la página virtual en la que se encuentra el código. Debido a que el tamaño de la página es 1 KB, la dirección virtual 1024 reside en la segunda página del espacio de direcciones virtuales (VPN=1, ya que VPN=0 es la primera página). Supongamos que esta página virtual se asigna al marco físico 4 (VPN 1 → PFN 4).

A continuación, está la matriz en sí. Su tamaño es de 4000 bytes (1000 enteros) y suponemos que reside en las direcciones virtuales 40000 a 44000 (sin incluir el último byte). Las páginas virtuales para este rango decimal son VPN=39...VPN=42. Por lo tanto, necesitamos asignaciones para estas páginas. Supongamos estas asignaciones de virtual a físico para el ejemplo: (VPN 39 → PFN 7), (VPN 40 → PFN 8), (VPN 41 → PFN 9), (VPN 42 → PFN 10).

---

<sup>2</sup>Aquí estamos haciendo un poco de trampa, asumiendo que cada instrucción tiene un tamaño de cuatro bytes para simplificar; En realidad, las instrucciones x86 tienen un tamaño variable.

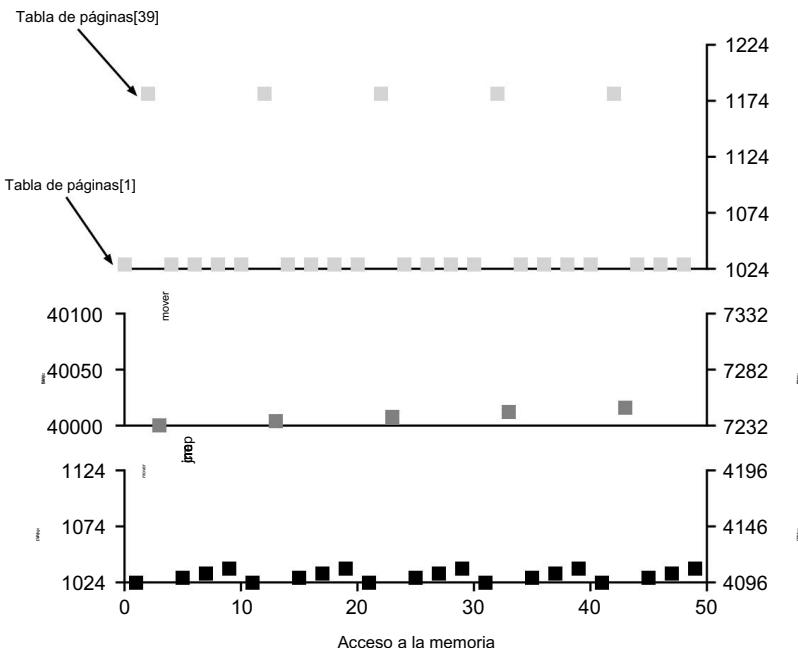


Figura 18.7: Un seguimiento de la memoria virtual (y física)

Ahora estamos listos para rastrear las referencias de memoria del programa.

Cuando se ejecuta, cada búsqueda de instrucción generará dos referencias de memoria: una a la tabla de páginas para encontrar el marco físico en el que reside la instrucción y otra a la instrucción misma para buscarla en la CPU para su procesamiento. Además, hay una referencia de memoria explícita en forma de instrucción `mov`; esto agrega primero otro acceso a la tabla de páginas (para traducir la dirección virtual de la matriz a la dirección física correcta) y luego el acceso a la matriz en sí.

El proceso completo, para las primeras cinco iteraciones del bucle, se muestra en la Figura 18.7 (página 11). El gráfico inferior muestra las referencias de la memoria de instrucciones en el eje y en negro (con las direcciones virtuales a la izquierda y las direcciones físicas reales a la derecha); el gráfico del medio muestra los accesos a la matriz en gris oscuro (nuevamente con el virtual a la izquierda y el físico a la derecha); Finalmente, el gráfico superior muestra los accesos a la memoria de la tabla de páginas en gris claro (solo físicos, ya que la tabla de páginas en este ejemplo reside en la memoria física). El eje x, para todo el seguimiento, muestra los accesos a la memoria en las primeras cinco iteraciones del bucle; Hay 10 accesos a la memoria por ciclo, que incluyen cuatro búsquedas de instrucciones, una actualización explícita de la memoria y cinco accesos a la tabla de páginas para traducir esas cuatro búsquedas y una actualización explícita.

Vea si puede encontrarle sentido a los patrones que aparecen en esta visualización. En particular, ¿qué cambiará a medida que el ciclo continúe ejecutándose?

¿Más allá de estas primeras cinco iteraciones? ¿Qué nuevas ubicaciones de memoria serán accedidos? ¿Puedes resolverlo?

Este ha sido sólo el ejemplo más simple (sólo unas pocas líneas de código C), y, sin embargo, es posible que ya pueda sentir la complejidad de comprender el comportamiento real de la memoria de aplicaciones reales. No os preocupéis: la situación empeora, porque los mecanismos que estamos a punto de introducir sólo complicar esta ya compleja maquinaria. ¡Lo siento!<sup>3</sup>

## 18.6 Resumen

Hemos introducido el concepto de paginación como una solución a nuestro desafío de virtualizar la memoria. La paginación tiene muchas ventajas sobre enfoques anteriores (como la segmentación). En primer lugar, no conduce a problemas externos.

fragmentación, ya que la paginación (por diseño) divide la memoria en tamaños fijos unidades. En segundo lugar, es bastante flexible y permite el uso escaso de espacios de direcciones virtuales.

Sin embargo, implementar soporte de paginación sin cuidado conducirá a un Máquina más lenta (con muchos accesos a memoria extra para acceder a la página). tabla), así como desperdicio de memoria (con la memoria llena de tablas de páginas en lugar de datos útiles de la aplicación). Por tanto, tendremos que pensar un poco más idear un sistema de localización que no sólo funcione, sino que funcione bien. Afortunadamente, los próximos dos capítulos nos mostrarán cómo hacerlo.

---

<sup>3</sup>Realmente no lo sentimos. Pero lamentamos no sentirlo, si eso tiene sentido.

## Referencias

[KE+62] "Sistema de almacenamiento de un nivel" por T. Kilburn, DBG Edwards, MJ Lanigan, FH Sumner. IRE Trans. EC-11, 2, 1962. Reimpreso en Bell y Newell, "Estructuras informáticas: lecturas y ejemplos". McGraw-Hill, Nueva York, 1971. El Atlas fue pionero en la idea de dividir la memoria en páginas de tamaño fijo y, en muchos sentidos, fue una forma temprana de las ideas de gestión de la memoria que vemos en los sistemas informáticos modernos.

[I09] "Manuales para desarrolladores de software de arquitecturas Intel 64 e IA-32" Intel, 2009. Disponible: <http://www.intel.com/products/processor/manuals>. En particular, preste atención al "Volumen 3A: Guía de programación del sistema, Parte 1" y al "Volumen 3B: Guía de programación del sistema, Parte 2".

[L78] "El Manchester Mark I y el Atlas: una perspectiva histórica" por SH Lavington. Communications of the ACM, Volumen 21:1, enero de 1978. Este artículo es una gran retrospectiva de parte de la historia del desarrollo de algunos sistemas informáticos importantes. Como a veces olvidamos en Estados Unidos, muchas de estas nuevas ideas vinieron del extranjero.

## Tarea (Simulación)

En esta tarea, utilizará un programa simple, conocido como `paging-linear-translate.py`, para ver si comprende cómo funciona la traducción simple de direcciones virtuales a físicas con tablas de páginas lineales. Consulte el archivo LÉAME para obtener más detalles.

Preguntas 1.

Antes de realizar cualquier traducción, usemos el simulador para estudiar cómo las tablas de páginas lineales cambian de tamaño según diferentes parámetros. Calcule el tamaño de las tablas de páginas lineales a medida que cambian los diferentes parámetros. Algunas aportaciones sugeridas se encuentran a continuación; Al usar el indicador `-v`, puede ver cuántas entradas de la tabla de páginas están llenas. Primero, para comprender cómo cambia el tamaño de la tabla de páginas lineal a medida que crece el espacio de direcciones, ejecute estas opciones:

```
-P 1k -a 1m -p 512m -v -n 0
-P 1k -a 2m -p 512m -v -n 0
-P 1k -a 4m -p 512m -v -n 0
```

Luego, para comprender cómo cambia el tamaño de la tabla de páginas lineal a medida que crece el tamaño de la página:

```
-P 1k -a 1m -p 512m -v -n 0
-P 2k -a 1m -p 512m -v -n 0
-P 4k -a 1m -p 512m -v -n 0
```

Antes de ejecutar cualquiera de estos, intente pensar en las tendencias esperadas. ¿Cómo debería cambiar el tamaño de la tabla de páginas a medida que crece el espacio de direcciones? ¿A medida que crece el tamaño de la página? ¿Por qué no utilizar páginas grandes en general?

2. Ahora hagamos algunas traducciones. Comience con algunos pequeños ejemplos y cambie la cantidad de páginas asignadas al espacio de direcciones con el indicador `-u`. Por ejemplo:

```
-P 1k -a 16k -p 32k -v -u 0
-P 1k -a 16k -p 32k -v -u 25
-P 1k -a 16k -p 32k -v -u 50
-P 1k -a 16k -p 32k -v -u 75
-P 1k -a 16k -p 32k -v -u 100
```

¿Qué sucede cuando aumenta el porcentaje de páginas asignadas en cada espacio de direcciones?

3. Ahora probemos algunas semillas aleatorias diferentes y algunos parámetros de espacio de direcciones diferentes (y a veces bastante locos), para variar:

```
-P 8 -a 32 -p 1024 -v -s 1  
-P 8k -a 32k -p 1m -v -s 2  
-P 1m -a 256m -p 512m -v -s 3
```

¿Cuáles de estas combinaciones de parámetros no son realistas? ¿Por qué?

4. Utilice el programa para probar otros problemas. ¿Puedes encontrar los límites donde el programa ya no funciona? Por ejemplo, ¿qué sucede si el tamaño del espacio de direcciones es mayor que la memoria física?

## Paginación: Mesas más pequeñas

Ahora abordamos el segundo problema que introduce la paginación: las tablas de páginas son demasiado grandes y, por tanto, consumen demasiada memoria. Comencemos con una tabla de páginas lineal. Como recordarás, las tablas de 1 página lineal se vuelven bonitas. grande. Supongamos nuevamente un espacio de dirección de  $32 \text{ bytes} = 32 \text{ bits} = 2^{32}$  y una entrada de tabla de páginas de 4 bytes. Por lo tanto, un espacio de direcciones tiene aproximadamente un millón de páginas  $\frac{2^{32}}{2^{12}} = 2^{20}$ ; multiplicar por el tamaño de entrada de la tabla de páginas virtuales (y verá que nuestra tabla de páginas tiene un tamaño de 4 MB. Recuerde también: ¡normalmente tenemos una tabla de páginas para cada proceso en el sistema! Con cien procesos activos (no es raro en como sistema moderno), asignaremos cientos de megabytes de memoria solo para tablas de páginas. Como resultado, estamos buscando algunas técnicas para reducir esta pesada carga. ¡Hay muchas, así que comencemos! antes de nuestro quid:

### CRUX: ¿CÓMO HACER LAS TABLAS DE PÁGINAS MÁS PEQUEÑAS?

Las tablas de páginas simples basadas en matrices (generalmente llamadas tablas de páginas lineales) son demasiado grandes y ocupan demasiada memoria en los sistemas típicos. ¿Cómo podemos hacer que las tablas de páginas sean más pequeñas? ¿Cuáles son las ideas clave? ¿Qué inefficiencies surgen como resultado de estas nuevas estructuras de datos?

### 20.1 Solución simple: páginas más grandes

Podríamos reducir el tamaño de la tabla de páginas de una forma sencilla: utilizar páginas más grandes. Tome nuevamente nuestro espacio de direcciones de 32 bits, pero esta vez suponga páginas de 16 KB. Tendríamos así una VPN de 18 bits más un offset de 14 bits. Suponiendo el mismo tamaño para cada PTE (4 bytes), ahora tenemos  $2^{18}$  entradas en 2 tablas de páginas lineales y, por lo tanto, un tamaño total de 1 MB por tabla de páginas, un factor

<sup>10</sup> quizás no; Esto de la localización se está saliendo de control, ¿no? Dicho esto, asegúrese siempre de comprender el problema que está resolviendo antes de pasar a la solución; de hecho, si comprende el problema, a menudo podrá encontrar la solución usted mismo. Aquí, el problema debería ser claro: las tablas de páginas lineales simples (basadas en matrices) son demasiado grandes.

## ADEMÁS: MÚLTIPLES TAMAÑOS DE

PÁGINA Además, tenga en cuenta que muchas arquitecturas (por ejemplo, MIPS, SPARC, x86-64) ahora admiten múltiples tamaños de página. Por lo general, se utiliza un tamaño de página pequeño (4 KB u 8 KB). Sin embargo, si una aplicación "inteligente" lo solicita, se puede utilizar una única página grande (por ejemplo, de 4 MB) para una porción específica del espacio de direcciones, lo que permite a dichas aplicaciones colocar una estructura de datos de uso frecuente (y grande) en dicho espacio mientras consume solo una única entrada TLB. Este tipo de uso de páginas grandes es común en los sistemas de gestión de bases de datos y otras aplicaciones comerciales de alto nivel. Sin embargo, la razón principal para utilizar varios tamaños de página no es ahorrar espacio en la tabla de páginas; es reducir la presión sobre el TLB, permitiendo que un programa acceda a más espacio de direcciones sin sufrir demasiadas fallas de TLB. Sin embargo, como han demostrado los investigadores [N+02], el uso de múltiples tamaños de página hace que el administrador de memoria virtual del sistema operativo sea notablemente más complejo y, por lo tanto, las páginas grandes a veces se usan más fácilmente simplemente exportando una nueva interfaz a las aplicaciones para solicitar páginas grandes directamente.

reducción de cuatro en el tamaño de la tabla de páginas (no es sorprendente que la reducción refleje exactamente el aumento de cuatro en el tamaño de la página).

Sin embargo, el principal problema de este enfoque es que las páginas grandes generan desperdicio dentro de cada página, un problema conocido como fragmentación interna (ya que el desperdicio es interno a la unidad de asignación). Por lo tanto, las aplicaciones terminan asignando páginas pero solo usando pequeños fragmentos de cada una, y la memoria se llena rápidamente con estas páginas demasiado grandes. Por lo tanto, la mayoría de los sistemas utilizan tamaños de página relativamente pequeños en el caso común: 4 KB (como en x86) u 8 KB (como en SPARCv9). Lamentablemente, nuestro problema no se resolverá tan fácilmente.

## 20.2 Enfoque híbrido: paginación y segmentos

Siempre que tengas dos enfoques razonables pero diferentes para algo en la vida, siempre debes examinar la combinación de los dos para ver si puedes obtener lo mejor de ambos mundos. A esta combinación la llamamos híbrido. Por ejemplo, ¿por qué comer sólo chocolate o mantequilla de maní cuando puedes combinar los dos en un hermoso híbrido conocido como Reese's Peanut Butter Cup [M28]?

Hace años, a los creadores de Multics (en particular a Jack Dennis) se les ocurrió esta idea al construir el sistema de memoria virtual Multics [M07]. Específicamente, Dennis tuvo la idea de combinar paginación y segmentación para reducir la sobrecarga de memoria de las tablas de páginas.

Podemos ver por qué esto podría funcionar examinando con más detalle una tabla de páginas lineal típica. Supongamos que tenemos un espacio de direcciones en el que las porciones utilizadas del montón y la pila son pequeñas. Por ejemplo, utilizamos un pequeño espacio de direcciones de 16 KB con páginas de 1 KB (Figura 20.1); la tabla de páginas para este espacio de direcciones se encuentra en la Figura 20.2.

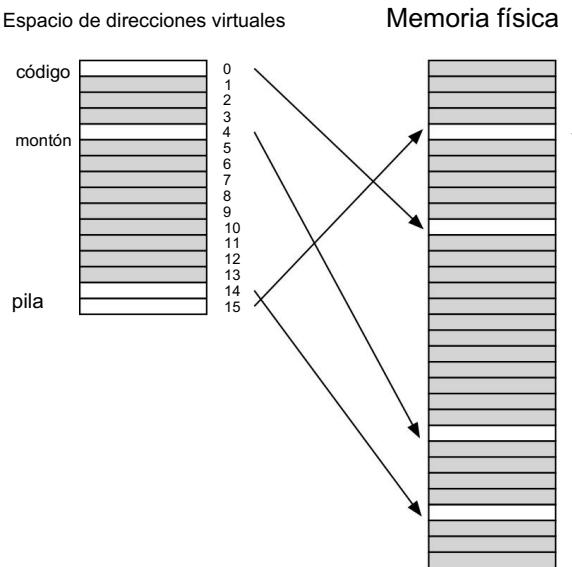


Figura 20.1: Un espacio de direcciones de 16 KB con páginas de 1 KB

PFN protección válida presente sucia			
10	1 1 0 — R& — - 0 — -	0	—
-	1 1 0 — - 0 — - 0 — -	—	—
-	0 — - 0 — - 0 — - 0	—	—
-	— - 0 — - 0 — - 1 1 1	—	—
23	1 rw-	1	—
-	—	—	—
-	—	—	—
-	—	—	—
-	—	—	—
-	—	—	—
-	—	—	—
-	—	—	—
28	rw-	1	—
4	rw-	1	—

Figura 20.2: Tabla de páginas para un espacio de direcciones de 16 KB

Este ejemplo supone que la página de códigos única (VPN 0) está asignada a la página física 10, la página de montón única (VPN 4) a la página física 23, y las dos páginas de pila en el otro extremo del espacio de direcciones (VPN 14 y

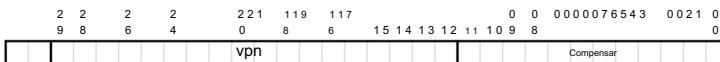
15) están asignados a las páginas físicas 28 y 4, respectivamente. Como puede ver en la imagen, la mayor parte de la tabla de páginas no se utiliza y está llena de entradas no válidas . ¡Qué pérdida! Y esto es para un pequeño espacio de direcciones de 16 KB. ¡Imagínese la tabla de páginas de un espacio de direcciones de 32 bits y todo el espacio potencial desperdiciado allí! En realidad, no imagines tal cosa; es demasiado espantoso.

De ahí nuestro enfoque híbrido: en lugar de tener una tabla de páginas única para todo el espacio de direcciones del proceso, ¿por qué no tener una por segmento lógico? En este ejemplo, podríamos tener tres tablas de páginas, una para las partes de código, montón y pila del espacio de direcciones.

Ahora, recuerde que con la segmentación teníamos un registro base que nos decía dónde vivía cada segmento en la memoria física y un registro límite que nos decía el tamaño de dicho segmento. En nuestro híbrido, todavía tenemos esas estructuras en la MMU; Aquí, usamos la base no para apuntar al segmento en sí, sino para contener la dirección física de la tabla de páginas de ese segmento. El registro de límites se utiliza para indicar el final de la tabla de páginas (es decir, cuántas páginas válidas tiene).

Hagamos un ejemplo sencillo para aclarar. Suponga un espacio de direcciones virtuales de 32 bits con páginas de 4 KB y un espacio de direcciones dividido en cuatro segmentos. Solo usaremos tres segmentos para este ejemplo: uno para código, otro para montón y otro para pila.

Para determinar a qué segmento se refiere una dirección, usaremos los dos bits superiores del espacio de direcciones. Supongamos que 00 es el segmento no utilizado, con 01 para el código, 10 para el montón y 11 para la pila. Por lo tanto, una dirección virtual se ve así: 3 3 2 2 2 2 1 0 7 5 3 2 Seg



En el hardware, supongamos que hay tres pares de base/límites, uno para código, otro para montón y otro para pila. Cuando se ejecuta un proceso, el registro base para cada uno de estos segmentos contiene la dirección física de una tabla de páginas lineales para ese segmento; por lo tanto, cada proceso en el sistema ahora tiene tres tablas de páginas asociadas. En un cambio de contexto, estos registros deben cambiarse para reflejar la ubicación de las tablas de páginas del proceso que se ejecuta recientemente.

En un fallo de TLB (asumiendo un TLB gestionado por hardware, es decir, donde el hardware es responsable de manejar los fallos de TLB), el hardware utiliza los bits de segmento (SN) para determinar qué par de base y límites usar. Luego, el hardware toma la dirección física que contiene y la combina con la VPN de la siguiente manera para formar la dirección de la entrada de la tabla de páginas (PTE):

```

SN           = (Dirección virtual y SEG_MASK) >> SN_SHIFT
vpn          = (Dirección virtual y VPN_MASK) >> VPN_SHIFT
DirecciónDePTE = Base[SN] + (VPN * tamañoDe(PTE))

```

Esta secuencia debería resultarle familiar; es prácticamente idéntico a lo que vimos antes con las tablas de páginas lineales. La única diferencia, por supuesto, es el uso de uno de los registros base de tres segmentos en lugar del registro base de la tabla de una sola página.

**CONSEJO: UTILIZA**

**HÍBRIDOS** Cuando tengas dos ideas buenas y aparentemente opuestas, siempre debes ver si puedes combinarlas en un híbrido que logre lograr lo mejor de ambos mundos. Se sabe que las especies híbridas de maíz, por ejemplo, son más robustas que cualquier especie natural. Por supuesto, no todos los híbridos son una buena idea; vea el Zeedonk (o Zonkey), que es un cruce de una cebra y un burro. Si no cree que exista una criatura así, búsquela y prepárese para sorprenderse.

La diferencia crítica en nuestro esquema híbrido es la presencia de un registro de límites por segmento; cada registro de límites contiene el valor de la página máxima válida en el segmento. Por ejemplo, si el segmento de código utiliza sus primeras tres páginas (0, 1 y 2), la tabla de páginas del segmento de código solo tendrá tres entradas asignadas y el registro de límites se establecerá en 3; Los accesos a la memoria más allá del final del segmento generarán una excepción y probablemente conducirán a la terminación del proceso. De esta manera, nuestro enfoque híbrido logra importantes ahorros de memoria en comparación con la tabla de páginas lineal; Las páginas no asignadas entre la pila y el montón ya no ocupan espacio en una tabla de páginas (solo para marcarlas como no válidas).

Sin embargo, como habrás notado, este enfoque no está exento de problemas. En primer lugar, todavía requiere que utilicemos la segmentación; Como comentamos antes, la segmentación no es tan flexible como nos gustaría, ya que supone un cierto patrón de uso del espacio de direcciones; Si tenemos un montón grande pero poco utilizado, por ejemplo, aún podemos terminar con una gran cantidad de desperdicio en la tabla de páginas. En segundo lugar, este híbrido hace que vuelva a surgir la fragmentación externa. Si bien la mayor parte de la memoria se administra en unidades del tamaño de una página, las tablas de páginas ahora pueden tener un tamaño arbitrario (en múltiplos de PTE). Por tanto, encontrar espacio libre para ellos en la memoria es más complicado. Por estas razones, la gente siguió buscando mejores formas de implementar tablas de páginas más pequeñas.

### 20.3 Tablas de páginas multinivel

Un enfoque diferente no se basa en la segmentación sino que ataca el mismo problema: ¿cómo deshacerse de todas esas regiones no válidas en la tabla de páginas en lugar de mantenerlas todas en la memoria? A este enfoque lo llamamos tabla de páginas de varios niveles, ya que convierte la tabla de páginas lineal en algo parecido a un árbol. Este enfoque es tan eficaz que muchos sistemas modernos lo emplean (por ejemplo, x86 [BOH10]). Ahora describimos este enfoque en detalle.

La idea básica detrás de una tabla de páginas de varios niveles es simple. Primero, divida la tabla de páginas en unidades del tamaño de una página; luego, si una página completa de entradas de la tabla de páginas (PTE) no es válida, no asigne esa página de la tabla de páginas en absoluto. Para rastrear si una página de la tabla de páginas es válida (y si es válida, dónde está en la memoria), use una nueva estructura, llamada directorio de páginas. Por lo tanto, el directorio de páginas se puede utilizar para indicarle dónde está una página de la tabla de páginas o que toda la página de la tabla de páginas no contiene páginas válidas.

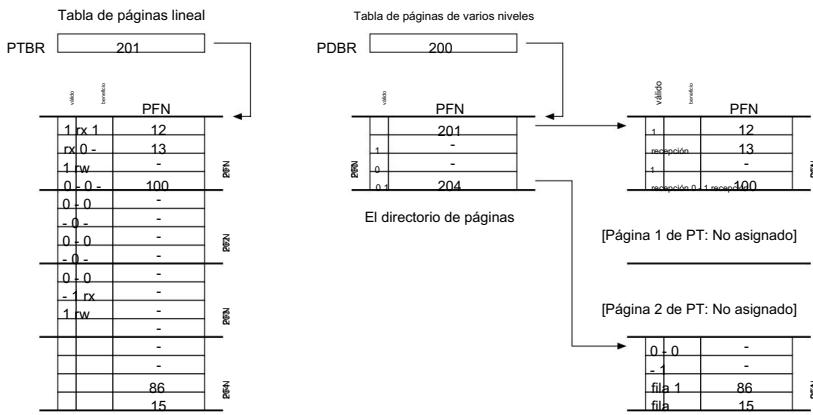


Figura 20.3: Tablas de páginas lineales (izquierda) y multinivel (derecha)

La figura 20.3 muestra un ejemplo. A la izquierda de la figura está el clásico tabla de páginas lineales; aunque la mayoría de las regiones medias de la dirección espacio no son válidos, aún requerimos espacio en la tabla de páginas asignado para esos regiones (es decir, las dos páginas centrales de la tabla de páginas). A la derecha hay un tabla de páginas de varios niveles. El directorio de páginas marca solo dos páginas del tabla de páginas reside en la memoria. Y así puedes ver una forma de visualizar Qué hace una tabla de varios niveles: simplemente forma partes de la página lineal. La tabla desaparece (liberando esos fotogramas para otros usos) y las pistas que la tabla asignan con el directorio de páginas.

El directorio de páginas, en una tabla simple de dos niveles, contiene una entrada por página de la tabla de páginas. Consta de varias entradas de directorio de páginas (PDE). Una PDE (como mínimo) tiene un bit válido y un número de marco de página (PFN), similar a un PTE. Sin embargo, como se indicó anteriormente, el significado de este bit válido es ligeramente diferente: si la PDE es válida, significa que al menos una de las páginas de la tabla de páginas a la que apunta la entrada (a través del PFN) es válida, es decir, en al menos una PTE en esa página señalada por esta PDE, la bit válido en ese PTE está establecido en uno. Si la PDE no es válida (es decir, igual a cero), el resto de la PDE no está definido.

Las tablas de páginas de varios niveles tienen algunas ventajas obvias sobre los enfoques hemos visto hasta ahora. En primer lugar, y quizás lo más obvio, la tabla de niveles múltiples sólo asigna espacio de tabla de páginas en proporción a la cantidad de direcciones. espacio que estás utilizando; por lo tanto, generalmente es compacto y admite espacios de dirección escasos.

En segundo lugar, si se construye cuidadosamente, cada parte de la tabla de páginas encaja claramente dentro de una página, lo que facilita la gestión de la memoria; el sistema operativo puede simplemente tomar la siguiente página libre cuando necesite asignar o hacer crecer una página

#### CONSEJO: COMPRENDA LAS COMPENSACIONES

**TIEMPO-ESPACIO** Al construir una estructura de datos, siempre se deben considerar las compensaciones tiempo-espacio en su construcción. Generalmente, si desea acelerar el acceso a una estructura de datos particular, tendrá que pagar una penalización por el uso de espacio de la estructura.

mesa. En contraste, esto con una tabla de páginas lineal simple (no paginada)<sup>2</sup>, cual es solo una matriz de PTE indexadas por VPN; Con tal estructura, toda la tabla de páginas lineal debe residir de forma contigua en la memoria física. Para una tabla de páginas grande (por ejemplo, 4 MB), encontrar una porción tan grande de memoria física contigua y libre sin utilizar puede ser todo un desafío. Con una estructura de varios niveles, agregamos un nivel de indirección mediante el uso del directorio de páginas, que apunta a partes de la tabla de páginas; esa dirección indirecta nos permite colocar páginas de tabla de páginas donde queramos en la memoria física.

Cabe señalar que las mesas de varios niveles tienen un costo; En caso de fallo de TLB, se necesitarán dos cargas de la memoria para obtener la información de traducción correcta de la tabla de páginas (una para el directorio de páginas y otra para el propio PTE), en contraste con una sola carga con una tabla de páginas lineal. Por tanto, la tabla de niveles múltiples es un pequeño ejemplo de una compensación tiempo-espacio. Queríamos mesas más pequeñas (y las conseguimos), pero no gratis; Aunque en el caso común (golpe de TLB), el rendimiento es obviamente idéntico, un fallo de TLB tiene un coste mayor con esta mesa más pequeña.

Otro aspecto negativo evidente es la complejidad. Ya sea que sea el hardware o el sistema operativo el que maneje la búsqueda de la tabla de páginas (en caso de fallo de TLB), hacerlo es sin duda más complicado que una simple búsqueda lineal de la tabla de páginas. A menudo estamos dispuestos a aumentar la complejidad para mejorar el rendimiento o reducir los gastos generales; en el caso de una tabla de varios niveles, complicamos las búsquedas en la tabla de páginas para ahorrar memoria valiosa.

Un ejemplo detallado de varios niveles Para

comprender mejor la idea detrás de las tablas de páginas de varios niveles, hagamos un ejemplo. Imagine un pequeño espacio de direcciones de 16 KB de tamaño, con páginas de 64 bytes. Así, tenemos un espacio de direcciones virtuales de 14 bits, con 8 bits para la VPN y 6 bits para el desplazamiento. Una tabla de páginas lineal tendría  $2^{14}$  (256<sup>8</sup>) entradas, incluso si solo se utiliza una pequeña porción del espacio de direcciones. La figura 20.4 (página 8) presenta un ejemplo de dicho espacio de direcciones.

En este ejemplo, las páginas virtuales 0 y 1 son para el código, las páginas virtuales 4 y 5 para el montón y las páginas virtuales 254 y 255 para la pila; el resto de las páginas del espacio de direcciones no se utilizan.

Para crear una tabla de páginas de dos niveles para este espacio de direcciones, comenzamos con nuestra tabla de páginas lineal completa y la dividimos en unidades del tamaño de una página. Recuerde que nuestra tabla completa (en este ejemplo) tiene 256 entradas; supongamos que cada PTE tiene 4 bytes

---

<sup>2</sup>Aquí estamos haciendo algunas suposiciones, es decir, que todas las tablas de páginas residen en su totalidad en memoria física (es decir, no se intercambian en disco); Pronto relajaremos esta suposición.

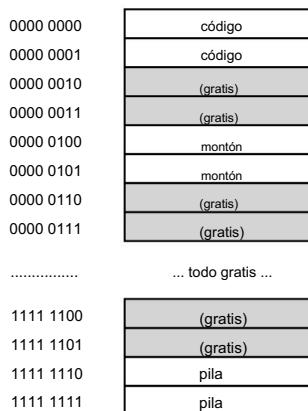
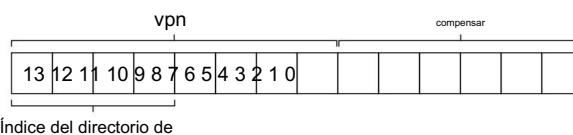


Figura 20.4: Un espacio de direcciones de 16 KB con páginas de 64 bytes

en tamaño. Por lo tanto, nuestra tabla de páginas tiene un tamaño de 1 KB ( $256 \times 4$  bytes). Dado que tenemos páginas de 64 bytes, la tabla de páginas de 1 KB se puede dividir en 16 páginas de 64 bytes; cada página puede contener 16 PTE.

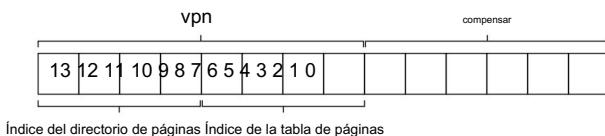
Lo que debemos entender ahora es cómo tomar una VPN y usarla para indexar primero en el directorio de páginas y luego en la página de la tabla de páginas. Recuerde que cada una es una serie de entradas; por lo tanto, todo lo que necesitamos descubrir es cómo construir el índice para cada uno de los componentes de la VPN.

Primero indexemos el directorio de la página. Nuestra tabla de páginas en este ejemplo es pequeña: 256 entradas, repartidas en 16 páginas. El directorio de páginas necesita una entrada por página de la tabla de páginas; por tanto, tiene 16 entradas. Como resultado, necesitamos cuatro bits de la VPN para indexar el directorio; Usamos los cuatro bits superiores de la VPN, de la siguiente manera:



**páginas** Una vez que extraemos el índice del directorio de páginas (PDIndex para abreviar) de la VPN, podemos usarlo para encontrar la dirección de la entrada del directorio de páginas (PDE) con un cálculo simple:  $PDEAddr = PageDirBase + (PDIndex * sizeof(PDE))$ . Esto da como resultado nuestro directorio de páginas, que ahora examinamos para seguir avanzando en nuestra traducción.

Si la entrada del directorio de páginas está marcada como no válida, sabemos que el acceso no es válido y, por lo tanto, generaremos una excepción. Sin embargo, si el PDE es válido, tenemos más trabajo por hacer. Específicamente, ahora tenemos que buscar la entrada de la tabla de páginas (PTE) de la página de la tabla de páginas a la que apunta esta entrada del directorio de páginas. Para encontrar esta PTE, tenemos que indexar la parte de la tabla de páginas usando los bits restantes de la VPN:



Este índice de tabla de páginas (PTIndex para abreviar) se puede utilizar para indexar en la propia tabla de páginas, dándonos la dirección de nuestro PTE:

$$\text{PTEAddr} = (\text{PDE.PFN} \ll \text{SHIFT}) + (\text{PTIndex} * \text{tamaño de (PTE)})$$

Tenga en cuenta que el número de marco de página (PFN) obtenido del directorio de páginas La entrada debe desplazarse hacia la izquierda antes de combinarla con el índice de la tabla de páginas para formar la dirección de la PTE.

Para ver si todo esto tiene sentido, ahora completaremos una tabla de páginas de varios niveles con algunos valores reales y traduciremos una única dirección virtual. vamos Comience con el directorio de páginas para este ejemplo (lado izquierdo de la Figura 20.5)

En la figura, puedes ver que cada entrada del directorio de páginas (PDF) describe algo

En este ejemplo, tenemos dos regiones válidas en el espacio de direcciones (en el rango de dirección 10.0.0.0 a 10.0.0.255).

En la página física 100 (el número de fotograma físico de la página 0 del tabla de páginas), tenemos la primera página de 16 entradas de la tabla de páginas para los primeros 16 VPN en el espacio de direcciones. Consulte la Figura 20.5 (parte central) para ver el contenido de esta parte de la tabla de páginas.

Esta página de la tabla de páginas contiene las asignaciones de los primeros 16 VPN. En nuestro ejemplo, las VPN 0 y 1 son válidas (el segmento de código), como

Directorio de páginas		Página de PT (@PFN:100)	Página de PT (@PFN:101)
¿PFN válido?	PFN válido	100	10 23
	1		rx—0—
— 0 recepción — 0 —		1 1	
— 0 — 0 — — 0 —			
— 0 — 0 — — 0 —			
— 0 rw- — 0 —		80	
— 0 rw- — 0 —	59		1 1
— 0 — 0 — — 0 —			
— 0 — 0 — — 0 —			
— 0 — 0 — — 0 —			
— 0 — 0 — — 0 —			
— 0 — 0 — — 0 —			
— 0 — 0 — — 0 —			
— 0 — 0 — — 0 —			
— 0 — 0 — — 0 —			
— 0 — 0 — — 0 —			
— 0 — 0 — — 0 —			
— 0 — 0 — — 0 —			
— 0 — 0 — 55 — 0 —	45		rw-
101	1		rw-
			1 1

Figura 20.5: Un directorio de páginas y partes de la tabla de páginas

## CONSEJO: TENGA CUIDADO CON LA

**COMPLEJIDAD** Los diseñadores de sistemas deben tener cuidado a la hora de añadir complejidad a su sistema. Lo que hace un buen constructor de sistemas es implementar el sistema menos complejo que logre la tarea en cuestión. Por ejemplo, si el espacio en disco es abundante, no debería diseñar un sistema de archivos que trabaje duro para utilizar la menor cantidad de bytes posible; de manera similar, si los procesadores son rápidos, es mejor escribir un módulo limpio y comprensible dentro del sistema operativo que quizás el código ensamblado a mano y optimizado para la CPU para la tarea en cuestión. Tenga cuidado con la complejidad innecesaria, en código optimizado prematuramente u otras formas; Estos enfoques hacen que los sistemas sean más difíciles de entender, mantener y depurar.

Como escribió Antoine de Saint-Exupéry: "La perfección finalmente se alcanza no cuando ya no hay nada que agregar, sino cuando ya no hay nada que quitar". Lo que no escribió: "Es mucho más fácil decir algo sobre la perfección que alcanzarla".

son 4 y 5 (el montón). Por lo tanto, la tabla tiene información cartográfica para cada una de esas páginas. El resto de entradas están marcadas como no válidas.

La otra página válida de la tabla de páginas se encuentra dentro de PFN 101. Esta página contiene asignaciones para las últimas 16 VPN del espacio de direcciones; consulte la Figura 20.5 (derecha) para obtener más detalles.

En el ejemplo, las VPN 254 y 255 (la pila) tienen asignaciones válidas.

Con suerte, lo que podemos ver en este ejemplo es cuánto ahorro de espacio es posible con una estructura indexada de varios niveles. En este ejemplo, en lugar de asignar las dieciséis páginas completas para una tabla de páginas lineal, asignamos solo tres: una para el directorio de páginas y dos para los fragmentos de la tabla de páginas que tienen asignaciones válidas. Los ahorros para espacios de direcciones grandes (32 o 64 bits) obviamente podrían ser mucho mayores.

Finalmente, usemos esta información para realizar una traducción.

Aquí hay una dirección que hace referencia al byte 0 de VPN 254: 0x3F80, o 11 1111 1000 0000 en binario.

Recuerde que usaremos los 4 bits superiores de la VPN para indexar el directorio de la página. Por lo tanto, 1111 elegirá la última entrada (la 15, si comienza en el 0) del directorio de páginas anterior. Esto nos dirige a una página válida de la tabla de páginas ubicada en la dirección 101. Luego usamos los siguientes 4 bits de la VPN (1110) para indexar esa página de la tabla de páginas y encontrar la PTE deseada. 1110 es la penúltima entrada (14.<sup>a</sup>) de la página y nos dice que la página 254 de nuestro espacio de direcciones virtuales está asignada a la página física 55. Concatenando PFN=55 (o hexadecimal 0x37) con offset=000000, podemos así formar nuestra dirección física deseada y emitir la solicitud al sistema de memoria: PhysAddr = (PTE.PFN << SHIFT) + offset  
= 00 1101 1100 0000 = 0x0DC0.

Ahora debería tener una idea de cómo construir una tabla de páginas de dos niveles, utilizando un directorio de páginas que apunte a las páginas de la tabla de páginas.

Lamentablemente, sin embargo, nuestro trabajo no ha terminado. Como veremos ahora, ¡a veces dos niveles de tabla de páginas no son suficientes!

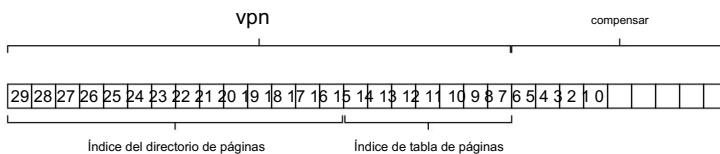
## Más de dos niveles

En nuestro ejemplo hasta ahora, hemos asumido que las tablas de páginas de varios niveles solo tienen dos niveles: un directorio de páginas y luego partes de la tabla de páginas. En algunos casos, es posible (y de hecho necesario) un árbol más profundo.

Tomemos un ejemplo simple y usémoslo para mostrar por qué una tabla multinivel más profunda puede ser útil. En este ejemplo, supongamos que tenemos un espacio de direcciones virtuales de 30 bits y una página pequeña (512 bytes). Por tanto, nuestra dirección virtual tiene un componente de número de página virtual de 21 bits y un desplazamiento de 9 bits.

Recuerde nuestro objetivo al construir una tabla de páginas de varios niveles: hacer que cada parte de la tabla de páginas encaje dentro de una sola página. Hasta ahora, sólo hemos considerado la tabla de páginas en sí; sin embargo, ¿qué pasa si el directorio de páginas se vuelve demasiado grande?

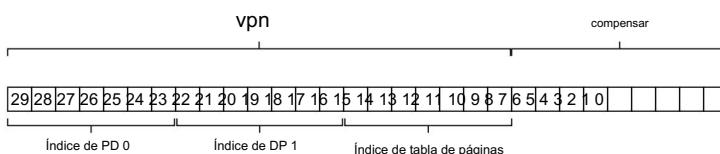
Para determinar cuántos niveles se necesitan en una tabla de varios niveles para que todas las partes de la tabla de páginas quepan dentro de una página, comenzamos determinando cuántas entradas de la tabla de páginas caben dentro de una página. Dado nuestro tamaño de página de 512 bytes y suponiendo un tamaño de PTE de 4 bytes, debería ver que puede colocar 128 PTE en una sola página. Cuando indexamos una página de la tabla de páginas, podemos concluir que necesitaremos los 7 bits menos significativos ( $\log_2 128$ ) de la VPN como índice:



Lo que también puedes notar en el diagrama anterior es cuántos bits se dejan en el directorio de páginas (grande): 14. Si nuestro directorio de páginas tiene 2 entradas, abarca no una página sino 128 y, por lo tanto, nuestro objetivo de hacer que cada pieza de la tabla de páginas de varios niveles encaje en una página desaparece. 14

Para remediar este problema, construimos un nivel adicional del árbol, dividiendo el directorio de páginas en varias páginas y luego agregando otro directorio de páginas encima, para que apunte a las páginas del directorio de páginas.

Así podemos dividir nuestra dirección virtual de la siguiente manera:



Ahora, al indexar el directorio de páginas de nivel superior, utilizamos los bits superiores de la dirección virtual (Índice PD 0 en el diagrama); este índice se puede utilizar para recuperar la entrada del directorio de páginas del directorio de páginas de nivel superior. Si es válido, el segundo nivel del directorio de páginas se consulta combinando el número de marco físico del PDE de nivel superior y el

```

1 VPN = (Dirección virtual y VPN_MASK) >> SHIFT
2 (Éxito, TlbEntry) = TLB_Lookup(VPN)
3 si (Éxito == Verdadero)           // Golpe TLB
4     si (CanAccess(TlbEntry.ProtectBits) == Verdadero)
5         Desplazamiento = Dirección virtual y OFFSET_MASK
6         PhysAddr = (TlbEntry.PFN << MAYÚS) | Compensar
7         Registro = AccessMemory(PhysAddr)
8     demás
9     RaiseException(PROTECTION_FAULT)
10 más //                         // Señorita TLB
11     primero, obtener la entrada del directorio de la página
12     PDIndex = (VPN y PD_MASK) >> PD_SHIFT
13     PDEAddr = PDBR + (PDIndex * tamaño de (PDE))
14     PDE      = MemoriadeAcceso(PDEAddr)
15     si (PDE.Válido == Falso)
16         RaiseException (SEGMENTATION_FAULT)
17     demás
18     // PDE es válido: ahora recupera PTE de la tabla de páginas
19     PTIndex = (VPN y PT_MASK) >> PT_SHIFT
20     PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * tamaño de (PTE))
21     PTE      = MemoriadeAcceso(PTEAddr)
22     si (PTE.Válido == Falso)
23         RaiseException (SEGMENTATION_FAULT)
24     de lo contrario si (CanAccess(PTE.ProtectBits) == Falso)
25         RaiseException(PROTECTION_FAULT)
26     demás
27     TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
28     Reintentar instrucción()

```

Figura 20.6: Flujo de control de tabla de páginas de varios niveles

siguiente parte de la VPN (índice PD 1). Finalmente, si es válida, la dirección PTE. se puede formar utilizando el índice de la tabla de páginas combinado con la dirección del PDE de segundo nivel. ¡Uf! Eso es mucho trabajo. Y todo solo para busque algo en una tabla de varios niveles.

#### El proceso de traducción: recuerde el TLB

Para resumir todo el proceso de traducción de direcciones utilizando un sistema de dos niveles tabla de páginas, presentamos una vez más el flujo de control en forma algorítmica (Figura 20.6). La figura muestra lo que sucede en el hardware (asumiendo una TLB administrado por hardware) en cada referencia de memoria.

Como puede ver en la figura, antes de que se produzca cualquier acceso complicado a la tabla de páginas de varios niveles, el hardware primero verifica el TLB; al un hit, la dirección física se forma directamente sin acceder a la página mesa, como antes. Sólo en caso de fallo de TLB es necesario que el hardware realice la búsqueda completa de varios niveles. En este camino, puedes ver el costo de nuestra tabla de páginas tradicional de dos niveles: dos accesos de memoria adicionales a busque una traducción válida.

## 20.4 Tablas de páginas invertidas

Un ahorro de espacio aún mayor en el mundo de las tablas de páginas se obtiene con las tablas de páginas invertidas. Aquí, en lugar de tener muchas tablas de páginas (una por proceso del sistema), mantenemos una tabla de páginas única que tiene una entrada para cada página física del sistema. La entrada nos dice qué proceso está utilizando esta página y qué página virtual de ese proceso se asigna a esta página física.

Encontrar la entrada correcta ahora es cuestión de buscar en esta estructura de datos. Un escaneo lineal sería costoso y, por lo tanto, a menudo se construye una tabla hash sobre la estructura base para acelerar las búsquedas. El PowerPC es un ejemplo de dicha arquitectura [JM98].

De manera más general, las tablas de páginas invertidas ilustran lo que hemos dicho desde el principio: las tablas de páginas son solo estructuras de datos. Puedes hacer muchas locuras con las estructuras de datos, haciéndolas más pequeñas o más grandes, más lentas o más rápidas. Las tablas de páginas invertidas y de varios niveles son sólo dos ejemplos de las muchas cosas que se pueden hacer.

## 20.5 Intercambio de tablas de páginas al disco

Finalmente, discutimos la relajación de un último supuesto. Hasta ahora, hemos asumido que las tablas de páginas residen en la memoria física propiedad del núcleo. Incluso con nuestros muchos trucos para reducir el tamaño de las tablas de páginas, aún es posible, sin embargo, que sean demasiado grandes para caber en la memoria todas a la vez. Por lo tanto, algunos sistemas colocan dichas tablas de páginas en la memoria virtual del núcleo, lo que permite que el sistema intercambie algunas de estas tablas de páginas al disco cuando la presión de la memoria es un poco escasa. Hablaremos más sobre esto en un capítulo futuro (es decir, el estudio de caso sobre VAX/VMS), una vez que comprendamos cómo mover páginas dentro y fuera de la memoria con más detalle.

## 20.6 Resumen Ahora

hemos visto cómo se construyen tablas de páginas reales; no necesariamente solo como matrices lineales sino como estructuras de datos más complejas. Las compensaciones que presentan estas tablas son en el tiempo y el espacio (cuanto más grande es la mesa, más rápido se puede solucionar un fallo de TLB, y viceversa) y, por lo tanto, la elección correcta de la estructura depende en gran medida de las limitaciones del entorno dado. ambiente.

En un sistema con memoria limitada (como muchos sistemas más antiguos), las estructuras pequeñas tienen sentido; En un sistema con una cantidad razonable de memoria y con cargas de trabajo que utilizan activamente una gran cantidad de páginas, una tabla más grande que acelere los errores de TLB podría ser la opción correcta. Con los TLB administrados por software, todo el espacio de las estructuras de datos se abre para el deleite del innovador del sistema operativo (pista: ese eres tú). ¿Qué nuevas estructuras se te ocurren? ¿Qué problemas resuelven? Piense en estas preguntas mientras se queda dormido y sueña los grandes sueños que sólo los desarrolladores de sistemas operativos pueden soñar.

## Referencias

- [BOH10] "Sistemas informáticos: la perspectiva de un programador" por Randal E. Bryant y David R. O'Hallaron. Addison-Wesley, 2010. Todavía tenemos que encontrar una buena primera referencia a la tabla de páginas de varios niveles. Sin embargo, este gran libro de texto de Bryant y O'Hallaron profundiza en los detalles de x86, que al menos es uno de los primeros sistemas que utilizaba dichas estructuras. También es un gran libro para tener.
- [JM98] "Memoria virtual: cuestiones de implementación" por Bruce Jacob, Trevor Mudge. IEEE Computer, junio de 1998. Un excelente estudio de varios sistemas diferentes y su enfoque para virtualizar la memoria. Muchos detalles sobre x86, PowerPC, MIPS y otras arquitecturas.
- [LL82] "Gestión de memoria virtual en el sistema operativo VAX/VMS" por Hank Levy, P. Lipman. Computadora IEEE, vol. 15, No. 3, marzo de 1982. Un artículo fantástico sobre un administrador de memoria virtual real en un sistema operativo clásico, VMS. De hecho, es tan fantástico que lo usaremos para repasar todo lo que hemos aprendido sobre la memoria virtual hasta ahora dentro de algunos capítulos.
- [M28] "Tazas de mantequilla de maní de Reese" de Mars Candy Corporation. Publicado en tiendas cercanas a usted. Aparentemente, estos finos dulces fueron inventados en 1928 por Harry Burnett Reese, un ex granjero lechero y capataz de envío de un tal Milton S. Hershey. Al menos eso es lo que dice en Wikipedia. De ser cierto, Hershey y Reese probablemente se odian mutuamente, como deberían hacerlo dos barones del chocolate.
- [N+02] "Soporte de sistema operativo práctico y transparente para Superpages" por Juan Navarro, Sitaram Iyer, Peter Druschel, Alan Cox. OSDI '02, Boston, Massachusetts, octubre de 2002. Un bonito artículo que muestra todos los detalles que hay que acertar para incorporar páginas grandes, o superpáginas, en un sistema operativo moderno. No es tan fácil como podría pensar, por desgracia.
- [M07] "Multics: Historia" Disponible: <http://www.multicians.org/history.html>. Este increíble sitio web proporciona una gran cantidad de historia sobre el sistema Multics, sin duda uno de los sistemas más influyentes en la historia del sistema operativo. La cita allí contenida: "Jack Dennis del MIT contribuyó con ideas arquitectónicas influyentes al comienzo de Multics, especialmente la idea de combinar paginación y segmentación". (de la Sección 1.2.1)

## Tarea (Simulación)

Esta pequeña y divertida tarea pone a prueba si comprendes cómo funciona una tabla de páginas de varios niveles. Y sí, existe cierto debate sobre el uso del término "diversión" en la frase anterior. El programa se llama, tal vez como era de esperar: paging-multilevel-translate.py; consulte el archivo LÉAME para obtener más detalles.

### Preguntas 1.

Con una tabla de páginas lineal, necesita un solo registro para ubicar la tabla de páginas, asumiendo que el hardware realiza la búsqueda cuando se pierde un TLB. ¿Cuántos registros necesitas para ubicar una tabla de páginas de dos niveles? ¿Una mesa de tres niveles?

2. Utilice el simulador para realizar traducciones dadas las semillas aleatorias 0, 1 y 2, y verifique sus respuestas usando el indicador -c . ¿Cuántas referencias de memoria se necesitan para realizar cada búsqueda?
3. Dado su conocimiento de cómo funciona la memoria caché, ¿cómo cree que se comportarán las referencias de memoria a la tabla de páginas en la memoria caché? ¿Conducirán a muchos aciertos de caché (y, por tanto, a accesos rápidos?) ¿O muchos fallos (y, por tanto, accesos lentos)?

## Concurrencia: una introducción

Hasta ahora, hemos visto el desarrollo de las abstracciones básicas que realiza el sistema operativo. Hemos visto cómo tomar una única CPU física y convertirla en múltiples CPU virtuales, permitiendo así la ilusión de múltiples programas ejecutándose al mismo tiempo. También hemos visto cómo crear la ilusión de una gran memoria virtual privada para cada proceso; esta abstracción del espacio de direcciones permite que cada programa se comporte como si tuviera su propia memoria cuando, en realidad, el sistema operativo está multiplexando secretamente espacios de direcciones en la memoria física (y, a veces, en el disco).

En esta nota, presentamos una nueva abstracción para un único proceso en ejecución: la de un hilo. En lugar de nuestra visión clásica de un único punto de ejecución dentro de un programa (es decir, una única PC desde donde se obtienen y ejecutan las instrucciones), un programa multiproceso tiene más de un punto de ejecución (es decir, múltiples PC, cada uno de los cuales se está recuperando y ejecutando). Quizás otra forma de pensar en esto es que cada hilo es muy parecido a un proceso separado, excepto por una diferencia: comparten el mismo espacio de direcciones y, por lo tanto, pueden acceder a los mismos datos.

Por tanto, el estado de un único hilo es muy similar al de un proceso. Tiene un contador de programa (PC) que rastrea de dónde obtiene el programa las instrucciones. Cada hilo tiene su propio conjunto privado de registros que utiliza para el cálculo; por lo tanto, si hay dos subprocesos que se ejecutan en un solo procesador, al pasar de ejecutar uno (T1) a ejecutar el otro (T2), se debe realizar un cambio de contexto . El cambio de contexto entre subprocesos es bastante similar al cambio de contexto entre procesos, ya que el estado de registro de T1 debe guardarse y el estado de registro de T2 debe restaurarse antes de ejecutar T2. Con los procesos, guardamos el estado en un bloque de control de procesos (PCB); ahora, necesitaremos uno o más bloques de control de subprocesos (TCB) para almacenar el estado de cada subproceso de un proceso. Sin embargo, hay una diferencia importante en el cambio de contexto que realizamos entre subprocesos en comparación con los procesos: el espacio de direcciones sigue siendo el mismo (es decir, no hay necesidad de cambiar qué tabla de páginas estamos usando).

Otra diferencia importante entre subprocesos y procesos tiene que ver con la pila. En nuestro modelo simple del espacio de direcciones de un proceso clásico.

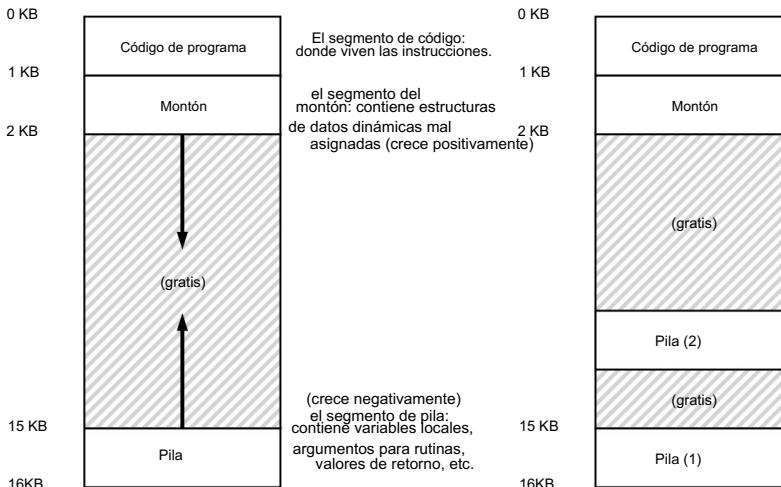


Figura 26.1: Espacios de direcciones de subproceso único y de subprocesos múltiples

(que ahora podemos llamar proceso de un solo subproceso), hay una única pila, que generalmente reside en la parte inferior del espacio de direcciones (Figura 26.1, izquierda).

Sin embargo, en un proceso de subprocesos múltiples, cada subproceso se ejecuta de forma independiente y, por supuesto, puede llamar a varias rutinas para realizar cualquier trabajo que esté realizando. En lugar de una única pila en el espacio de direcciones, habrá una por hilo. Digamos que tenemos un proceso de subprocesos múltiples que tiene dos subprocesos; el espacio de direcciones resultante se ve diferente (Figura 26.1, derecha).

En esta figura, puede ver dos pilas repartidas por el espacio de direcciones del proceso. Por lo tanto, cualquier variable, parámetro, valor de retorno y otras cosas asignadas a la pila que coloquemos en la pila se colocarán en lo que a veces se llama almacenamiento local de subprocesos, es decir, la pila del subproceso relevante.

También puedes notar cómo esto arruina nuestro hermoso diseño del espacio de direcciones. Antes, la pila y el montón podían crecer de forma independiente y los problemas sólo surgían cuando se quedaba sin espacio en el espacio de direcciones. Aquí ya no tenemos una situación tan agradable. Afortunadamente, esto suele estar bien, ya que las pilas generalmente no tienen que ser muy grandes (la excepción es en programas que hacen un uso intensivo de la recursividad).

## 26.1 ¿Por qué utilizar subprocesos?

Antes de entrar en detalles sobre los subprocesos y algunos de los problemas que podría tener al escribir programas multiproceso, primero respondamos una pregunta más simple. ¿Por qué debería utilizar hilos?

Resulta que hay al menos dos razones principales por las que debería utilizar trampas. La primera es simple: el paralelismo. Imagine que está escribiendo un programa que realiza operaciones en matrices muy grandes, por ejemplo, agregando dos matrices grandes juntas, o incrementando el valor de cada elemento en la matriz en cierta cantidad. Si está ejecutando un solo procesador, la tarea es sencilla: simplemente realice cada operación y listo.

Sin embargo, si está ejecutando el programa en un sistema con múltiples procesadores, tiene el potencial de acelerar este proceso considerablemente utilizando los procesadores para que cada uno realice una parte del trabajo. El trabajo de transformar su programa estándar de un solo subproceso en un programa que realice este tipo de trabajo en múltiples CPU se llama paralelización, y usar un subproceso por CPU para realizar este trabajo es una opción natural y típica. forma de hacer que los programas se ejecuten más rápido en hardware moderno.

La segunda razón es un poco más sutil: para evitar bloquear el programa. progreso debido a una E/S lenta. Imagine que está escribiendo un programa que realiza diferentes tipos de E/S: ya sea esperando para enviar o recibir un mensaje, para que se complete una E/S de disco explícita o incluso (implícitamente) para una página. falta para terminar. En lugar de esperar, es posible que su programa desee hacer algo más, incluido utilizar la CPU para realizar cálculos, o incluso emitir más solicitudes de E/S. Usar hilos es una forma natural de evitar quedarse atascado; mientras un hilo en su programa espera (es decir, está bloqueado esperando E/S), el programador de la CPU puede cambiar a otros subprocesos, lo que están listos para ejecutarse y hacer algo útil. El enhebrado permite la superposición de E/S con otras actividades dentro de un solo programa, de manera muy similar a como lo hizo la multiprogramación para procesos entre programas; Como resultado, muchos modernos aplicaciones basadas en servidor (servidores web, sistemas de gestión de bases de datos, y similares) hacen uso de subprocesos en sus implementaciones.

Por supuesto, en cualquiera de los casos mencionados anteriormente, podría utilizar varios procesos en lugar de subprocesos. Sin embargo, los hilos comparten un espacio de direcciones y, por lo tanto, facilitar el intercambio de datos y, por lo tanto, son una opción natural cuando construir este tipo de programas. Los procesos son una opción más sensata para tareas lógicamente separadas donde se necesita poco compartir estructuras de datos en la memoria.

## 26.2 Un ejemplo: creación de hilos

Entremos en algunos de los detalles. Digamos que queremos ejecutar un programa que crea dos hilos, cada uno de los cuales realiza un trabajo independiente, en este caso imprimiendo "A" o "B". El código se muestra en la Figura 26.2 (página 4).

El programa principal crea dos subprocesos, cada uno de los cuales ejecutará el función `mythread()`, aunque con argumentos diferentes (la cadena A o B). Una vez que se crea un hilo, puede comenzar a ejecutarse de inmediato (dependiendo de según los caprichos del planificador); alternativamente, se puede poner en estado "listo" pero estado no "en ejecución" y, por lo tanto, aún no se está ejecutando. Por supuesto, en un multiprocesador, Los hilos podrían incluso estar ejecutándose al mismo tiempo, pero no nos preocupemos sobre esta posibilidad todavía.

```

1 #incluir <stdio.h>
2 #incluir <afirmar.h>
3 #incluir <pthread.h>
4 #incluir "común.h"
5 #incluir "common_threads.h"

6
7 void *mythread(void *arg) {
8     printf("%s\n", (char *) arg);
9     devolver NULO;
10    }
11
12 enteros
13 principal(int argc, char *argv[]) {
14     pthread_t p1, p2;
15     intrc;
16     printf("principal: comenzar\n");
17     Pthread_create(&p1, NULL, mito, "A");
18     Pthread_create(&p2, NULL, mito, "B");
19     // unirse espera a que terminen los hilos
20     Pthread_join(p1, NULO);
21     Pthread_join(p2, NULO);
22     printf("principal: fin\n");
23     devolver 0;
24 }
```

Figura 26.2: Código de creación de hilo simple (t0.c)

Después de crear los dos hilos (llamémoslos T1 y T2), el principal hilo llama a `pthread join()`, que espera a que se inicie un hilo en particular. El hilo hace dos veces, asegurando así que T1 y T2 se ejecuten y completen completamente. Lo hace dos veces, asegurando así que T1 y T2 se ejecuten y completen antes de permitir finalmente que el hilo principal se ejecute nuevamente; cuando lo haga, imprimirá "principal: fin" y saldrá. En total, se emplearon tres hilos.

durante esta ejecución: el hilo principal, T1 y T2.

Examinemos el posible orden de ejecución de este pequeño programa. En el diagrama de ejecución (Figura 26.3, página 5), el tiempo aumenta en dirección descendente y cada columna muestra cuándo se activa un hilo diferente (el principal, o el subproceso 1, o el subproceso 2) se está ejecutando.

Tenga en cuenta, sin embargo, que este orden no es el único orden posible. De hecho, dada una secuencia de instrucciones, hay bastantes, dependiendo de qué hilo el programador decide ejecutar en un punto determinado. Por ejemplo, Una vez que se crea un hilo, puede ejecutarse inmediatamente, lo que llevaría a la ejecución se muestra en la Figura 26.4 (página 5).

También podríamos ver "B" impresa antes de "A", si, por ejemplo, el programador decidió ejecutar el subproceso 2 primero a pesar de que el subproceso 1 se creó antes; no hay razón para suponer que un hilo que se crea primero se ejecutará primero.

La Figura 26.5 (página 6) muestra este orden de ejecución final, con el Hilo 2 llegando a pavonearse antes del Hilo 1.

Como podrá ver, una forma de pensar sobre la creación de hilos

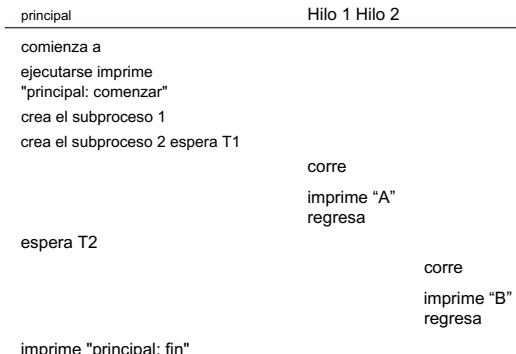


Figura 26.3: Trazado del hilo (1)

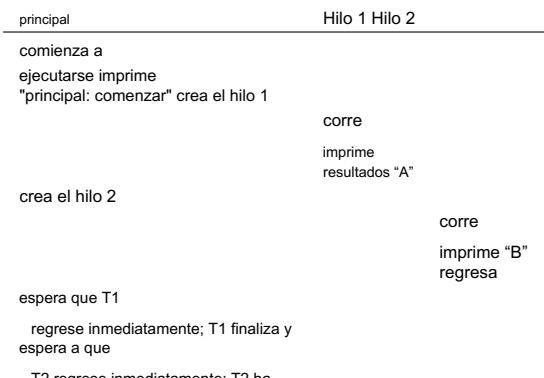


Figura 26.4: Trazado del hilo (2)

es que es un poco como hacer una llamada a función; sin embargo, en lugar de ejecutar primero la función y luego regresar a la persona que llama, el sistema crea un nuevo hilo de ejecución para la rutina que se está llamando y se ejecuta independientemente de la persona que llama, tal vez antes de regresar de la creación. , pero quizás mucho más tarde. Lo que se ejecuta a continuación lo determina el programador del sistema operativo y, aunque es probable que el programador implemente algún algoritmo sensato, es difícil saber qué se ejecutará en un momento dado.

Como también podrás ver en este ejemplo, los subprocessos complican la vida: ¡ya es difícil saber qué se ejecutará y cuándo! Las computadoras son bastante difíciles de entender sin concurrencia. Desafortunadamente, con la concurrencia, la cosa simplemente empeora. Mucho peor.

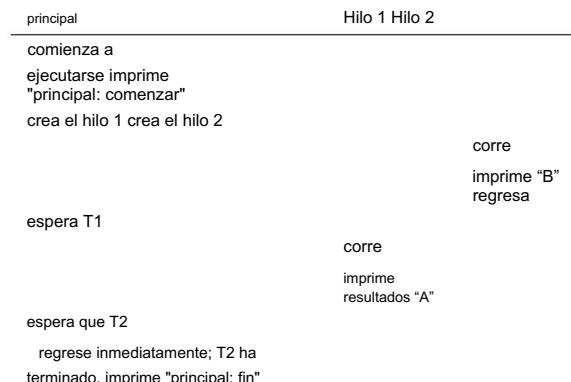


Figura 26.5: Trazado del hilo (3)

### 26.3 Por qué empeora: datos compartidos El ejemplo de hilo

simple que mostramos arriba fue útil para mostrar cómo se crean los hilos y cómo se pueden ejecutar en diferentes órdenes dependiendo de cómo el programador decide ejecutarlos. Sin embargo, lo que no muestra es cómo interactúan los subprocessos cuando acceden a datos compartidos.

Imaginemos un ejemplo simple donde dos hilos desean actualizar un variable global compartida. El código que estudiaremos se encuentra en la Figura 26.6 (página 7).

Aquí hay algunas notas sobre el código. Primero, como sugiere Stevens [SR05], ajustamos la creación del hilo y unimos rutinas para simplemente salir en caso de falla; Para un programa tan simple como este, al menos queremos notar que ocurrió un error (si es que ocurrió), pero no hacer nada muy inteligente al respecto (por ejemplo, simplemente salir). Por lo tanto, Pthread create() simplemente llama a pthread create() y se asegura de que el código de retorno sea 0; si no es así, Pthread create() simplemente imprime un mensaje y sale.

En segundo lugar, en lugar de usar dos cuerpos de función separados para los subprocessos de trabajo, simplemente usamos una sola pieza de código y le pasamos al subprocesso un argumento (en este caso, una cadena) para que podamos hacer que cada subprocesso imprima una letra diferente antes. sus mensajes.

Finalmente, y lo más importante, ahora podemos ver lo que cada trabajador intenta hacer: agregar un número al contador de variables compartidas y hacerlo 10 millones de veces ( $1e7$ ) en un bucle. Por tanto, el resultado final deseado es: 20.000.000.

Ahora compilamos y ejecutamos el programa para ver cómo se comporta. Algunas veces, todo funciona como podríamos esperar: indicador> gcc -o main main.c -Wall -pthread; ./main principal: comenzar (contador = 0)

```

A: comenzar B:
comenzar A:
terminar B:
terminar
principal: terminar con ambos (contador = 20000000)

```

```

1 #incluir <stdio.h>
2 #incluir <pthread.h>
3 #incluir "común.h"
4 #incluir "common_threads.h"
5
6 contador int volátil estático = 0;
7
8 // hilo mito()
9 //
10 // Simplemente agrega 1 al contador repetidamente, en un bucle
11 // No, así no es como sumarías 10.000.000 a
12 // un contador, pero muestra muy bien el problema.
13 //
14 void *mythread(void *arg) {
15     printf("%s: comenzar\n", (char *) arg);
16     ent yo;
17     para (yo = 0; yo < 1e7; yo++) {
18         contador = contador + 1;
19     }
20     printf("%s: hecho\n", (char *) arg);
21     devolver NULO;
22 }
23
24 // principal()
25 //
26 // Simplemente lanza dos hilos (pthread_create)
27 // y luego los espera (pthread_join)
28 //
29 int principal(int argc, char *argv[]) {
30     pthread_t p1, p2;
31     printf("principal: comenzar (contador = %d)\n", contador);
32     Pthread_create(&p1, NULL, mito, "A");
33     Pthread_create(&p2, NULL, mito, "B");
34
35     // unirse espera a que terminen los hilos
36     Pthread_join(p1, NULO);
37     Pthread_join(p2, NULO);
38     printf("principal: hecho con ambos (contador = %d)\n",
39             encimera);
40     devolver 0;
41 }

```

Figura 26.6: Compartir datos: Uh Oh (t1.c)

Desafortunadamente, cuando ejecutamos este código, incluso en un solo procesador, no necesariamente obtiene el resultado deseado. A veces obtenemos:

```
indicador> ./main
principal: comenzar (contador = 0)
A: comenzar B:
comenzar A:
hecho B:
hecho
principal: hecho con ambos (contador = 19345221)
```

Intentémoslo una vez más, sólo para ver si nos hemos vuelto locos. Después de todo, ¿no se supone que las computadoras producen resultados deterministas , como te han enseñado? ¿Quizás tus profesores te han estado mintiendo? (jadear)

```
indicador> ./main
principal: comenzar (contador = 0)
A: comenzar B:
comenzar A:
hecho B:
hecho
principal: hecho con ambos (contador = 19221041)
```

¡No sólo cada ejecución es incorrecta, sino que también produce un resultado diferente! Queda una gran pregunta: ¿por qué sucede esto?

#### CONSEJO: CONOCE Y UTILIZA TUS HERRAMIENTAS

Siempre debes aprender nuevas herramientas que te ayuden a escribir, depurar y comprender los sistemas informáticos. Aquí utilizamos una elegante herramienta llamada desensamblador. Cuando ejecuta un desensamblador en un ejecutable, le muestra qué instrucciones de ensamblaje componen el programa. Por ejemplo, si deseamos comprender el código de bajo nivel para actualizar un contador (como en nuestro ejemplo), ejecutamos objdump (Linux) para ver el código ensamblador: indicador>

```
objdump -d main Al hacerlo, se genera
```

una lista larga de todos las instrucciones del programa, claramente etiquetadas (especialmente si compiló con el indicador -g), que incluyen información de símbolos en el programa. El programa objdump es sólo una de las muchas herramientas que debería aprender a utilizar; un depurador como gdb, perfiladores de memoria como valgrind o purify y, por supuesto, el compilador en sí son otros sobre los que debería dedicar tiempo para aprender más; cuanto mejor utilice sus herramientas, mejores sistemas podrá construir.

## 26.4 El meollo del problema: programación incontrolada

Para comprender por qué sucede esto, debemos comprender la secuencia de código que genera el compilador para que la actualización contrarreste. En este caso, simplemente deseamos agregar un número (1) al contador. Por lo tanto, la secuencia de código para hacerlo podría verse así (en x86):

```
mov 0x8049a1c, %eax agrega
$0x1, %eax mov %eax,
0x8049a1c
```

Este ejemplo supone que el contador de variables está ubicado en la dirección 0x8049a1c. En esta secuencia de tres instrucciones, la instrucción x86 mov se usa primero para obtener el valor de la memoria en la dirección y colocarlo en el registro eax. Luego, se realiza la suma, sumando 1 (0x1) al contenido del registro eax y, finalmente, el contenido de eax se almacena nuevamente en la memoria en la misma dirección.

Imaginemos que uno de nuestros dos subprocesos (Subproceso 1) ingresa a esta región de código y, por lo tanto, está a punto de incrementar el contador en uno. Carga el valor del contador (digamos que es 50 para empezar) en su registro eax. Por lo tanto, eax=50 para el subproceso 1. Luego agrega uno al registro; por lo tanto eax=51.

Ahora sucede algo desafortunado: se activa una interrupción del temporizador; por lo tanto, el sistema operativo guarda el estado del hilo que se está ejecutando actualmente (su PC, sus registros, incluido eax, etc.) en el TCB del hilo.

Ahora sucede algo peor: se elige el subproceso 2 para ejecutarlo e ingresa este mismo fragmento de código. También ejecuta la primera instrucción, obteniendo el valor del contador y colocándolo en su eax (recuerde: cada subproceso cuando se ejecuta tiene sus propios registros privados; los registros se virtualizan mediante el código de cambio de contexto que los guarda y restaura). El valor del contador sigue siendo 50 en este punto y, por lo tanto, el subproceso 2 tiene eax=50. Entonces supongamos que el subproceso 2 ejecuta las siguientes dos instrucciones, incrementando eax en 1 (por lo tanto eax=51) y luego guardando el contenido de eax en el contador (dirección 0x8049a1c). Por tanto, el contador de la variable global ahora tiene el valor 51.

Finalmente, se produce otro cambio de contexto y el subproceso 1 continúa ejecutándose. Recuerde que acababa de ejecutar mov y add, y ahora está a punto de realizar la instrucción mov final. Recuerde también que eax=51. Por tanto, la instrucción mov final se ejecuta y guarda el valor en la memoria; el contador vuelve a ponerse en 51.

En pocas palabras, lo que sucedió es esto: el código para incrementar el contador se ejecutó dos veces, pero el contador, que comenzó en 50, ahora solo es igual a 51. Una versión "correcta" de este programa debería haber dado como resultado la variable contador igual a 52.

Veamos un seguimiento de ejecución detallado para comprender mejor el problema. Supongamos, para este ejemplo, que el código anterior se carga en la dirección 100 de la memoria, como la siguiente secuencia (nota para aquellos que están acostumbrados a bonitos conjuntos de instrucciones tipo RISC: x86 tiene instrucciones de longitud variable; esta instrucción mov ocupa 5 bytes de memoria y solo agrega 3):

SO	Hilo 1 antes	Hilo 2	(después de la instrucción) contador eax de pc
	de la sección crítica mov 8049a1c,%eax agrega \$0x1,%eax interrumpe		100 0 50 105 50 50 108 51 50
guardar T1			100 0 50
restaurar T2			105 50 50
interrumpir		mov 8049a1c, %eax 105 50 50 agregar \$0x1,%eax mov %eax,8049a1c 113 51 51	108 51 50
guardar T2			108 51 51
restaurar T1			113 51 51
movimiento %eax,8049a1c			

Figura 26.7: El problema: de cerca y en persona

100 mov	0x8049a1c, %eax
105 sumar	\$0x1, %eax
108 mov	%eax, 0x8049a1c

Con estos supuestos, lo que sucede se muestra en la Figura 26.7 (página 10). Suponga que el contador comienza en el valor 50 y siga este ejemplo para asegurarse de que comprende lo que está pasando.

Lo que hemos demostrado aquí se llama condición de carrera (o, más específicamente, una carrera de datos): los resultados dependen del tiempo de ejecución de el código. Con algo de mala suerte (es decir, cambios de contexto que ocurren en puntos inoportunos de la ejecución), obtenemos un resultado incorrecto. De hecho, podemos obtener un resultado diferente cada vez; por lo tanto, en lugar de un agradable cálculo determinista (al que estamos acostumbrados en las computadoras), llamamos a este resultado indeterminado, donde no se sabe cuál será el resultado y de hecho es probablemente sea diferente entre ejecuciones.

Debido a que varios subprocessos que ejecutan este código pueden generar una condición de carrera, llamamos a este código una sección crítica. Una sección crítica es una pieza de código que accede a una variable compartida (o más generalmente, a un recurso compartido) y no debe ser ejecutado simultáneamente por más de un subprocesso.

Lo que realmente queremos para este código es lo que llamamos exclusión mutua. Esta propiedad garantiza que si un subprocesso se ejecuta dentro del período crítico sección, los demás quedarán impedidos de hacerlo.

Por cierto, prácticamente todos estos términos fueron acuñados por Edsger Dijkstra, quien fue un pionero en este campo y de hecho ganó el Premio Turing, por este y otros trabajos; véase su artículo de 1968 sobre “Procesos secuenciales cooperativos” [D68] para obtener una descripción sorprendentemente clara del problema. Escucharemos más sobre Dijkstra en esta sección del libro.

**CONSEJO: UTILICE OPERACIONES**

**ATÓMICAS** Las operaciones atómicas son una de las técnicas subyacentes más poderosas en la construcción de sistemas informáticos, desde la arquitectura informática hasta el código concurrente (lo que estamos estudiando aquí) y los sistemas de archivos (que estudiaremos pronto). , sistemas de gestión de bases de datos e incluso sistemas distribuidos [L+93].

La idea detrás de hacer atómica una serie de acciones se expresa simplemente con la frase “todo o nada”; Debería aparecer como si todas las acciones que desea agrupar ocurrieran, o como si no ocurriera ninguna, sin ningún estado intermedio visible. En ocasiones, la agrupación de muchas acciones en una única acción atómica se denomina transacción, idea desarrollada con gran detalle en el mundo de las bases de datos y el procesamiento de transacciones [GR92].

En nuestro tema de exploración de la concurrencia, usaremos primitivas de sincronización para convertir secuencias cortas de instrucciones en bloques atómicos de ejecución, pero la idea de atomicidad es mucho más amplia que eso, como veremos. Por ejemplo, los sistemas de archivos utilizan técnicas como el registro en diario o la copia en escritura para realizar una transición atómica de su estado en el disco, fundamental para funcionar correctamente ante fallas del sistema. Si eso no tiene sentido, no se preocupe: lo tendrá en algún capítulo futuro.

## 26.5 El deseo de atomicidad

Una forma de resolver este problema sería tener instrucciones más poderosas que, en un solo paso, hicieran exactamente lo que necesitábamos y así eliminaran la posibilidad de una interrupción inoportuna. Por ejemplo, ¿qué pasaría si tuviéramos una super instrucción que se viera así?

agregar memoria 0x8049a1c, \$0x1

Supongamos que esta instrucción agrega un valor a una ubicación de memoria y el hardware garantiza que se ejecuta de forma atómica; cuando se ejecutara la instrucción, realizaría la actualización como se deseara. No se puede interrumpir a mitad de una instrucción, porque esa es precisamente la garantía que recibimos del hardware: cuando ocurre una interrupción, o la instrucción no se ha ejecutado en absoluto o se ha ejecutado hasta su finalización; no hay ningún estado intermedio.

El hardware puede ser algo hermoso, ¿no?

Atómicamente, en este contexto, significa “como una unidad”, lo que a veces entendemos como “todo o nada”. Lo que nos gustaría es ejecutar la secuencia de tres instrucciones de forma atómica:

```
mov 0x8049a1c, %eax agrega
$0x1, %eax mov %eax,
0x8049a1c
```

Como dijimos, si tuviéramos una sola instrucción para hacer esto, podríamos simplemente emitir esa instrucción y listo. Pero en el caso general no tendremos tal instrucción. Imaginemos que estábamos construyendo un árbol B concurrente y deseábamos actualizarlo; ¿Realmente querríamos que el hardware admitiera una instrucción de "actualización atómica del árbol B"? Probablemente no, al menos en un conjunto de instrucciones sensato.

Por lo tanto, lo que haremos será pedirle al hardware algunas instrucciones útiles sobre las cuales podamos construir un conjunto general de lo que llamamos primitivas de sincronización. Al utilizar este soporte de hardware, en combinación con algo de ayuda del sistema operativo, podremos crear código multiproceso que acceda a secciones críticas de manera sincronizada y controlada y, por lo tanto, produzca de manera confiable el resultado correcto a pesar de la naturaleza desafiante de ejecución concurrente. Bastante impresionante, ¿verdad?

Este es el problema que estudiaremos en esta sección del libro. Es un problema maravilloso y difícil, y debería hacerte doler la mente (un poco).

Si no es así, ¡entonces no lo entiendes! Sigue trabajando hasta que te duela la cabeza; entonces sabrás que vas en la dirección correcta. En ese momento, tómate un descanso; No queremos que te duela demasiado la cabeza.

#### EL CRUX: CÓMO SOPORTAR LA SINCRONIZACIÓN ¿Qué soporte

necesitamos del hardware para construir primitivas de sincronización útiles? ¿Qué soporte necesitamos del sistema operativo?

¿Cómo podemos construir estas primitivas de manera correcta y eficiente? ¿Cómo pueden los programas utilizarlos para obtener los resultados deseados?

## 26.6 Un problema más: esperando otro

Este capítulo ha planteado el problema de la concurrencia como si sólo ocurriera un tipo de interacción entre subprocesos, el de acceder a variables compartidas y la necesidad de soportar la atomicidad para las secciones críticas. Resulta que surge otra interacción común, donde un hilo debe esperar a que otro complete alguna acción antes de continuar. Esta interacción surge, por ejemplo, cuando un proceso realiza una E/S de disco y se pone en suspensión; Cuando se completa la E/S, es necesario despertar al proceso de su letargo para que pueda continuar.

Por lo tanto, en los próximos capítulos, no solo estudiaremos cómo construir soporte para primitivas de sincronización para soportar la atomicidad, sino también mecanismos para soportar este tipo de interacción entre dormir y despertar que es común en programas multiproceso. Si esto no tiene sentido ahora, ¡está bien! Lo será muy pronto, cuando lea el capítulo sobre variables de condición. Si para entonces no es así, entonces no está bien y deberías leer ese capítulo una y otra vez hasta que tenga sentido.

APARTE: TÉRMINOS CLAVE DE CONCURRENCIA  
 SECCIÓN CRÍTICA , CONDICIÓN DE CARRERA ,  
 INDETERMINADO, EXCLUSIÓN MUTUA Estos cuatro

términos son tan centrales para el código concurrente que pensamos que valía la pena mencionarlos explícitamente. Vea algunos de los primeros trabajos de Dijkstra [D65,D68] para más detalles.

- Una sección crítica es un fragmento de código que accede a un recurso compartido, generalmente una variable o estructura de datos.
- Una condición de carrera (o carrera de datos [NM92]) surge si múltiples subprocessos de ejecución ingresan a la sección crítica aproximadamente al mismo tiempo; ambos intentan actualizar la estructura de datos compartida, lo que lleva a un resultado sorprendente (y quizás indeseable).
- Un programa indeterminado consta de una o más condiciones de carrera; la salida del programa varía de una ejecución a otra, dependiendo de qué subprocessos se ejecutaron y cuándo. Por tanto, el resultado no es determinista, algo que normalmente esperamos de los sistemas informáticos.
- Para evitar estos problemas, los hilos deberían utilizar algún tipo de primitivas de exclusión mutua ; Al hacerlo, se garantiza que solo un subprocesso ingrese a una sección crítica, evitando así carreras y dando como resultado resultados deterministas del programa.

## 26.7 Resumen: ¿Por qué en la clase OS?

Antes de terminar, una pregunta que quizás tengas es: ¿por qué estudiamos esto en la clase de SO? "Historia" es la respuesta de una sola palabra; El sistema operativo fue el primer programa concurrente y se crearon muchas técnicas para su uso dentro del sistema operativo. Más tarde, con los procesos multiproceso, los programadores de aplicaciones también tuvieron que considerar estas cosas.

Por ejemplo, imagine el caso en el que hay dos procesos en ejecución. Supongamos que ambos llaman a write() para escribir en el archivo y ambos desean agregar los datos al archivo (es decir, agregar los datos al final del archivo, aumentando así su longitud). Para hacerlo, ambos deben asignar un nuevo bloque, registrar en el inodo del archivo donde reside este bloque y cambiar el tamaño del archivo para reflejar el nuevo tamaño más grande (entre otras cosas; aprenderemos más sobre los archivos en el tercera parte del libro). Debido a que una interrupción puede ocurrir en cualquier momento, el código que actualiza estas estructuras compartidas (por ejemplo, un mapa de bits para asignación o el inodo del archivo) son secciones críticas; por lo tanto, los diseñadores del sistema operativo, desde el comienzo de la introducción de la interrupción, tuvieron que preocuparse por cómo el sistema operativo actualiza las estructuras internas. Una interrupción intempestiva causa todos los problemas descritos anteriormente. No es de extrañar que para que funcionen correctamente sea necesario acceder cuidadosamente a las tablas de páginas, listas de procesos, estructuras de sistemas de archivos y prácticamente a todas las estructuras de datos del núcleo, con las primitivas de sincronización adecuadas.

## Referencias

[D65] "Solución de un problema en el control de programación concurrente" por EW Dijkstra. Communications of the ACM, 8(9):569, septiembre de 1965. Considerado el primer artículo de Dijkstra donde describe el problema de la exclusión mutua y una solución. La solución, sin embargo, no se utiliza mucho; Se necesita soporte avanzado de hardware y sistema operativo, como veremos en los próximos capítulos.

[D68] "Procesos secuenciales cooperativos" por Edsger W. Dijkstra. 1968. Disponible en este sitio: <http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF>. Dijkstra tiene una cantidad asombrosa de sus antiguos artículos, notas y pensamientos registrados (para la posteridad) en este sitio web en el último lugar donde trabajó, la Universidad de Texas. Sin embargo, gran parte de su trabajo fundamental lo realizó años antes, mientras estaba en la Technische Technische Hogeschool Eindhoven (THE), incluido este famoso artículo sobre "procesos secuenciales cooperativos", que básicamente describe todo el pensamiento que se necesita para escribir textos múltiples.

-Programas rosados. Dijkstra descubrió gran parte de esto mientras trabajaba en un sistema operativo que lleva el nombre de su escuela: el sistema operativo "EL" (decía "T", "H", "E", y no como la palabra "el").

[GR92] "Procesamiento de transacciones: conceptos y técnicas" por Jim Gray y Andreas Reuter. Morgan Kaufmann, septiembre de 1992. Este libro es la biblia del procesamiento de transacciones, escrito por una de las leyendas del campo, Jim Gray. Por este motivo también se considera el "volcado de cerebros" de Jim Gray, en el que anotó todo lo que sabe sobre cómo funcionan los sistemas de gestión de bases de datos. Lamentablemente, Gray falleció trágicamente hace unos años y muchos de nosotros perdimos a un amigo y gran mentor, incluidos los coautores de dicho libro, quienes tuvieron la suerte de interactuar con Gray durante sus años de posgrado.

[L+93] "Transacciones atómicas" de Nancy Lynch, Michael Merritt, William Weihl, Alan Fekete. Morgan Kaufmann, agosto de 1993. Un bonito texto sobre la teoría y la práctica de las transacciones atómicas para sistemas distribuidos. Quizás un poco formal para algunos, pero aquí se encuentra mucho buen material.

[NM92] "¿Qué son las condiciones de carrera? Algunas cuestiones y formalizaciones" de Robert HB Netzer y Barton P. Miller. ACM Letters on Programming Languages and Systems, Volumen 1:1, marzo de 1992. Una excelente discusión sobre los diferentes tipos de carreras que se encuentran en programas concurrentes. En este capítulo (y los siguientes), nos centramos en las carreras de datos, pero más adelante ampliaremos el tema para analizar también las carreras generales .

[SR05] "Programación avanzada en el entorno UNIX" por W. Richard Stevens y Stephen A. Rago. Addison-Wesley, 2005. Como hemos dicho muchas veces, compre este libro y léalo en pequeños trozos, preferiblemente antes de acostarse. De esta manera, te quedarás dormido más rápidamente; Más importante aún, aprenderá un poco más sobre cómo convertirse en un programador UNIX serio .

## Tarea (Simulación)

Este programa, x86.py, le permite ver cómo diferentes intercalaciones de subprocessos causan o evitan condiciones de carrera. Consulte el archivo README para obtener detalles sobre cómo funciona el programa y luego responda las preguntas a continuación.

### Preguntas

1. Examinemos un programa simple, "loop.s". Primero, simplemente lea y desactive entenderlo. Luego, ejecútelo con estos argumentos (./x86.py -p loop.s -t 1 -i 100 -R dx) Esto especifica un solo hilo, una interrupción cada 100 instrucciones y seguimiento del registro %dx. ¿Qué será %dx? estar durante la carrera? Utilice la opción -c para comprobar sus respuestas; las respuestas, a la izquierda, muestran el valor del registro (o valor de la memoria) después de que se haya ejecutado la instrucción de la derecha.
2. Mismo código, diferentes indicadores: (./x86.py -p loop.s -t 2 -i 100 -a dx=3,dx=3 -R dx) Esto especifica dos subprocessos e inicializa cada uno %dx a 3. ¿Qué valores verá %dx? Ejecute con -c para comprobar. Hace La presencia de múltiples subprocessos afecta sus cálculos? Hay una carrera en este código?
3. Ejecute esto: ./x86.py -p loop.s -t 2 -i 3 -r -a dx=3,dx=3 -R dx Esto hace que el intervalo de interrupción sea pequeño/aleatorio; use diferentes semillas (-s) para ver diferentes entrelazados. ¿La interrupción La frecuencia cambia algo?
4. Ahora, un programa diferente, looping-race-nolock.s, que accede a una variable compartida ubicada en la dirección 2000; llamaremos a esta variable valor. Ejecútelo con un solo hilo para confirmar su comprensión: ./x86.py -p looping-race-nolock.s -t 1 -M 2000 ¿Qué es el valor (es decir, en la dirección de memoria 2000) durante la ejecución? Utilice -c para comprobar.
5. Ejecutar con múltiples iteraciones/hilos: ./x86.py -p looping-race-nolock.s -t 2 -a bx=3 -M 2000 ¿Por qué Cada hilo se repite tres veces? ¿Cuál es el valor final del valor?
6. Ejecute con intervalos de interrupción aleatorios: ./x86.py -p bucle-carrera-nolock.s -t 2 -M 2000 -i 4 -r -s 0 con diferentes semillas (-s 1, -s 2, etc.) ¿Puedes saberlo mirando las Hilos entrelazados ¿Cuál será el valor final del valor? ¿El El momento de la interrupción es importante? ¿Dónde puede ocurrir de manera segura? Dónde no? En otras palabras, ¿dónde está exactamente la sección crítica?

7. Ahora examine los intervalos de interrupción fijos: ./x86.py -p looping-race-nolock.s -a bx=1 -t 2 -M 2000 -i 1 ¿Cuál será el valor final del valor de la variable compartida? ¿Qué pasa cuando cambias -i 2, -i 3, etc.? ¿Para qué intervalos de interrupción el programa da la respuesta "correcta"?
8. Ejecute lo mismo para más bucles (por ejemplo, establezca -a bx=100). ¿Qué intervalos de interrupción (-i) conducen a un resultado correcto? ¿Qué intervalos son sorprendentes?
9. Un último programa: espérame. Ejecutar: ./x86.py -p wait-for-me.s -a ax=0,ax=1 -R ax -M 2000 Esto establece el registro %ax en 1 para el hilo 0 y en 0 para el hilo 1, y vigila %ax y la ubicación de memoria 2000. ¿Cómo? ¿Debería comportarse el código? ¿Cómo utilizan los subprocesos el valor en la ubicación 2000? ¿Cuál será su valor final?
10. Ahora cambie las entradas: ./x86.py -p wait-for-me.s -a ax=0,ax=1 -R ax -M 2000 ¿Cómo se comportan los hilos? ¿Qué está haciendo el hilo 0? ¿Cómo cambiaría el resultado del seguimiento cambiando el intervalo de interrupción (por ejemplo, -i 1000, o quizás usar intervalos aleatorios)? ¿El programa utiliza eficientemente la CPU?

## Interludio: API de subprocessos

Este capítulo cubre brevemente las partes principales de la API de subprocessos. Cada parte se explicará con más detalle en los capítulos siguientes, a medida que mostremos cómo utilizar la API. Se pueden encontrar más detalles en varios libros y en línea.

fuentes [B89, B97, B+96, K+96]. Debemos tener en cuenta que los capítulos siguientes introducen los conceptos de bloqueos y variables de condición más lentamente, con muchos ejemplos; Por lo tanto, es mejor utilizar este capítulo como referencia.

### CRUX : CÓMO CREAR Y CONTROLAR HILOS

¿Qué interfaces debería presentar el sistema operativo para la creación y el control de subprocessos?  
 ¿Cómo deberían diseñarse estas interfaces para permitir la facilidad de uso y la utilidad?

## 27.1 Creación de hilos

Lo primero que debe poder hacer para escribir un programa multiproceso es crear nuevos subprocessos y, por lo tanto, debe existir algún tipo de interfaz de creación de subprocessos. En POSIX, es fácil:

```
#incluye <pthread.h> int
pthread_create(pthread_t *thread, const
               pthread_attr_t *attr, void *(*start_routine)(void*), void
               *arg);
```

Esta declaración puede parecer un poco compleja (especialmente si no ha usado punteros de función en C), pero en realidad no es tan mala. Hay cuatro argumentos: hilo, atributo, rutina de inicio y argumento. El primero, thread, es un puntero a una estructura de tipo pthread t; Usaremos esta estructura para interactuar con este hilo y, por lo tanto, debemos pasarl a pthread create() para inicializarlo.

El segundo argumento, attr, se utiliza para especificar cualquier atributo de este hilo. podría tener. Algunos ejemplos incluyen establecer el tamaño de la pila o quizás información sobre la prioridad de programación del hilo. Un atributo se inicializa con una llamada separada a pthread\_attr\_init(); consulte la página del manual para obtener más detalles. Sin embargo, en la mayoría de los casos, los valores predeterminados estarán bien; en este caso, simplemente pasaremos el valor NULL .

El tercer argumento es el más complejo, pero en realidad solo pregunta: ¿en qué función debería comenzar a ejecutarse este hilo? En C, llamamos a esto un puntero de función, y éste nos dice que se espera lo siguiente: un nombre de función (rutina de inicio), a la que se le pasa un único argumento de tipo void \* (como se indica entre paréntesis después de la rutina de inicio), y que devuelve un valor de tipo void \* (es decir, un puntero nulo).

Si esta rutina requiriera un argumento entero, en lugar de un puntero nulo, la declaración se vería así: int pthread\_create(..., // los

dos primeros argumentos son iguales void \*(\*start\_routine)(int), int arg) ;

Si, en cambio, la rutina tomó un puntero nulo como argumento, pero devolvió un número entero, se vería así:

```
int pthread_create(..., // los dos primeros argumentos son iguales int (*start_routine)(void
    *, void *arg);
```

Finalmente, el cuarto argumento, arg, es exactamente el argumento que se pasará a la función donde el hilo comienza a ejecutarse. Quizás se pregunte: ¿por qué necesitamos estos punteros vacíos? Bueno, la respuesta es bastante simple: tener un puntero vacío como argumento para la rutina de inicio de función nos permite pasar cualquier tipo de argumento; tenerlo como valor de retorno permite que el hilo devuelva cualquier tipo de resultado.

Veamos un ejemplo en la Figura 27.1. Aquí simplemente creamos un hilo al que se le pasan dos argumentos, empaquetados en un solo tipo que definimos nosotros mismos (myarg t). El hilo, una vez creado, puede simplemente convertir su argumento al tipo que espera y así descomprimir los argumentos como deseé.

¡Y ahí está! Una vez que crea un hilo, realmente tiene otra entidad de ejecución en vivo, completa con su propia pila de llamadas, ejecutándose dentro del mismo espacio de direcciones que todos los hilos existentes actualmente en el programa.

¡Así comienza la diversión!

## 27.2 Finalización del hilo

El ejemplo anterior muestra cómo crear un hilo. Sin embargo, ¿qué sucede si desea esperar a que se complete un hilo? Debes hacer algo especial para esperar a que finalice; en particular, debes llamar a la rutina pthread\_join().

---

```
int pthread_join(pthread_t hilo, void **value_ptr);
```

```

1 #incluir <stdio.h>
2 #incluir <pthread.h>
3
4 estructura typedef {
5     ent a;
6     intb;
7 } myarg_t;
8
9 void *mythread(void *arg) {
10     myarg_t *args = (myarg_t *) arg;
11     printf("%d %d\n", args->a, args->b);
12     devolver NULO;
13 }
14
15 int principal(int argc, char *argv[]) {
16     pthread_tp;
17     myarg_t argumentos = { 10, 20 };
18
19     int rc = pthread_create(&p, NULL, mythread, &args);
20     ...
21 }
```

Figura 27.1: Creación de un hilo

Esta rutina requiere dos argumentos. El primero es de tipo pthread t, y se utiliza para especificar qué hilo esperar. Esta variable es inicializada por la rutina de creación de hilos (cuando le pasas un puntero como argumento para pthread crear()); Si lo mantienes disponible, puedes usarlo para esperar ese hilo para terminar.

El segundo argumento es un puntero al valor de retorno que espera obtener. Atrás. Debido a que la rutina puede devolver cualquier cosa, se define para devolver un puntero a anular; porque la rutina pthread join()-cambia el valor del argumento pasado, debe pasar un puntero a ese valor, no sólo el valor en sí.

Veamos otro ejemplo (Figura 27.2, página 4). En el código, un Se vuelve a crear un único hilo y se le pasan un par de argumentos a través del estructura myarg t . Para devolver valores, se utiliza el tipo myret t . Una vez el hilo ha terminado de ejecutarse, el hilo principal, que ha estado esperando dentro de la rutina pthread join() 1 , luego regresa, y podemos acceder los valores devueltos por el hilo, es decir, lo que esté en myret t .

Algunas cosas a tener en cuenta sobre este ejemplo. Primero, muchas veces no Tenemos que hacer todo este doloroso empaquetar y desempaquetar argumentos. Para Por ejemplo, si simplemente creamos un hilo sin argumentos, podemos pasar NULL como argumento cuando se crea el hilo. De manera similar, podemos pasar NULL en pthread join() si no nos importa el valor de retorno.

---

<sup>1</sup>Tenga en cuenta que aquí utilizamos funciones contenedoras; específicamente, llamamos a Malloc(), Pthread join() y Pthread create(), que simplemente llama a sus versiones en minúsculas con nombres similares y se asegura de que

```

1 estructura typedef {int a; intb;} myarg_t;
2 estructura typedef {int x; int y;} myret_t;
3
4 void *mythread(void *arg) {
5     myret_t *rvals = Malloc(tamañode(myret_t));
6     valores->x = 1;
7     valores->y = 2;
8     return (void *) rvals;
9 }
10
11 int principal(int argc, char *argv[]) {
12     pthread_tp;
13     myret_t *rvals;
14     myarg_t argumentos = { 10, 20 };
15     Pthread_create(&p, NULL, mythread, &args);
16     Pthread_join(p, (void **) &rvals);
17     printf("devuelto %d %d\n", rvals->x, rvals->y);
18     libre(rvalls);
19     devolver 0;
20 }
```

Figura 27.2: Esperando a que se complete el hilo

En segundo lugar, si simplemente pasamos un valor único (por ejemplo, un valor largo, largo) int), no tenemos que empaquetarlo como un argumento. Figura 27.3 (página 5) muestra un ejemplo. En este caso, la vida es un poco más sencilla, ya que no tenemos que empaqueta argumentos y valores de retorno dentro de estructuras.

En tercer lugar, debemos señalar que hay que tener mucho cuidado con la forma en que los valores se devuelven desde un hilo. Específicamente, nunca devuelva un puntero que se refiere a algo asignado en la pila de llamadas del hilo. Si lo haces, ¿Qué crees que pasará? (¡piénselo!) A continuación se muestra un ejemplo de fragmento de código peligroso, modificado del ejemplo de la Figura 27.2.

```

1 void *mythread(void *arg) {
2     myarg_t *args = (myarg_t *) arg;
3     printf("%d %d\n", args->a, args->b);
4     myret_t Ups; // ASIGNADO EN LA PILA: ¡MALO!
5     Ups.x = 1;
6     Ups.y = 2;
7     devolver (nulo *) &oops;
8 }
```

En este caso, la variable Ups se asigna en la pila de Mythread. Sin embargo, cuando regresa, el valor se desasigna automáticamente (eso es ¡por qué la pila es tan fácil de usar, después de todo!), y por lo tanto, devolver un puntero

---

Las rutinas no arrojaron nada inesperado.

```

void *mythread(void *arg) {
    valor int largo largo = (int largo largo) arg; printf("%lld\n", valor); retorno
    (nulo *) (valor + 1);

}

int principal(int argc, char *argv[]) {
    pthread_tp; largo
    largo intervalo;
    Pthread_create(&p, NULL, mythread, (void *) 100);
    Pthread_join(p, (void **) &rvalue); printf("devuelto %lld\n",
    rvalue); devolver 0;

}

```

Figura 27.3: Argumento más simple pasando a un hilo

a una variable ahora desasignada conducirá a todo tipo de malos resultados. Ciertamente, cuando imprima los valores que cree haber devuelto, probablemente (¡pero no necesariamente!) se sorprenderá. ¡Pruébalo y descúbrelo tú mismo!

Finalmente, puedes notar que el uso de `pthread create()` para crear un hilo, seguido de una llamada inmediata a `pthread join()`, es una forma bastante extraña de crear un hilo. De hecho, existe una manera más sencilla de realizar exactamente esta tarea; se llama llamada a procedimiento. Claramente, normalmente crearemos más de un hilo y esperaremos a que se complete; de lo contrario, no tiene mucho sentido utilizar hilos.

Debemos tener en cuenta que no todo el código multiproceso utiliza la rutina de unión. Por ejemplo, un servidor web multiproceso podría crear varios subprocessos de trabajo y luego utilizar el hilo principal para aceptar solicitudes y pasárselas a los trabajadores, de forma indefinida. Por lo tanto, es posible que no sea necesario adherirse a programas de larga duración. Sin embargo, un programa paralelo que crea subprocessos para ejecutar una tarea particular (en paralelo) probablemente usará `join` para asegurarse de que todo ese trabajo se complete antes de salir o pasar a la siguiente etapa de cálculo.

### 27.3 Cerraduras

Más allá de la creación y unión de subprocessos, probablemente el siguiente conjunto de funciones más útil proporcionado por la biblioteca de subprocessos POSIX son aquellos para proporcionar exclusión mutua a una sección crítica mediante bloqueos. El par de rutinas más básico que se puede utilizar para este propósito lo proporciona lo siguiente:

```

int pthread_mutex_lock(pthread_mutex_t *mutex); int
pthread_mutex_unlock(pthread_mutex_t *mutex);

```

---

<sup>2</sup>Afortunadamente, el compilador gcc probablemente se quejará cuando escribas código como este, lo cual Es otra razón más para prestar atención a las advertencias del compilador.

Las rutinas deben ser fáciles de entender y utilizar. Cuando tiene una región de código que es una sección crítica y, por lo tanto, necesita protección para garantizar un funcionamiento correcto, los bloqueos son bastante útiles. Probablemente puedas imaginar cómo se ve el código:

```
bloqueo pthread_mutex_t;
pthread_mutex_lock(&bloqueo); x = x + 1; // o
cualquiera que sea su sección crítica pthread_mutex_unlock(&lock);
```

La intención del código es la siguiente: si ningún otro hilo mantiene el bloqueo cuando se llama a `pthread mutex lock()`, el hilo adquirirá el bloqueo y entrará en la sección crítica. Si otro hilo efectivamente mantiene el candado, el hilo que intenta capturar el candado no regresará de la llamada hasta que haya adquirido el candado (lo que implica que el hilo que sostiene el candado lo ha liberado mediante la llamada de desbloqueo). Por supuesto, muchos subprocessos pueden quedarse atrapados esperando dentro de la función de adquisición de bloqueo en un momento dado; Sin embargo, sólo el hilo con el bloqueo adquirido debe llamar a desbloquear.

Desafortunadamente, este código está roto en dos formas importantes. El primer problema es la falta de una inicialización adecuada. Todos los bloqueos deben inicializarse correctamente para garantizar que tengan los valores correctos para empezar y, por lo tanto, funcionen como se desea cuando se llaman a bloquear y desbloquear.

Con subprocessos POSIX, hay dos formas de inicializar bloqueos. De una sola mano Para hacer esto es usar PTHREAD\_MUTEX\_INITIALIZER, de la siguiente manera:

```
pthread_mutex_t bloqueo = PTHREAD_MUTEX_INITIALIZER;
```

Al hacerlo, se establece el bloqueo en los valores predeterminados y, por lo tanto, se puede utilizar el bloqueo. La forma dinámica de hacerlo (es decir, en tiempo de ejecución) es realizar una llamada a `pthread mutex init()`, de la siguiente manera:

```
int rc = pthread_mutex_init(&bloqueo, NULL); afirmar(rc == 0); // ¡siempre
comprueba el éxito!
```

El primer argumento de esta rutina es la dirección de la cerradura misma, mientras que el segundo es un conjunto opcional de atributos. Lea más sobre los atributos usted mismo; pasar NULL simplemente usa los valores predeterminados. Cualquiera de las dos formas funciona, pero normalmente utilizamos el método dinámico (último). Tenga en cuenta que también se debe realizar una llamada correspondiente a `pthread mutex destroy()` cuando haya terminado con el bloqueo; consulte la página del manual para obtener todos los detalles.

El segundo problema con el código anterior es que no verifica los códigos de error al llamar a bloquear y desbloquear. Al igual que prácticamente cualquier rutina de biblioteca que usted llama en un sistema UNIX, ¡estas rutinas también pueden fallar! Si su código no verifica correctamente los códigos de error, la falla ocurrirá silenciosamente, lo que en este caso podría permitir que varios subprocessos ingresen a una sección crítica. Como mínimo, utilice envoltorios que afirman que la rutina tuvo éxito, como se muestra en la Figura 27.4 (página 7); Los programas más sofisticados (que no son juguetes), que no pueden simplemente salir cuando algo sale mal, deben verificar si hay fallas y hacer algo apropiado cuando una llamada no tiene éxito.

```
// Mantiene el código limpio; usar solo si exit() OK en caso de falla void
Pthread_mutex_lock(pthread_mutex_t *mutex) { int rc = pthread_mutex_lock(mutex);
    afirmar(rc == 0);

}
```

Figura 27.4: Un contenedor de ejemplo

Las rutinas de bloqueo y desbloqueo no son las únicas rutinas dentro del Biblioteca pthreads para interactuar con cerraduras. Otras dos rutinas de interés:

```
int pthread_mutex_trylock(pthread_mutex_t *mutex); int
pthread_mutex_timedlock(pthread_mutex_t *mutex, estructura timespec *abs_timeout);
```

Estas dos llamadas se utilizan en la adquisición de cerraduras. La versión trylock devuelve un error si el bloqueo ya está mantenido; la versión timedlock para adquirir un bloqueo regresa después de un tiempo de espera o después de adquirir el bloqueo, lo que ocurra primero. Por tanto, el timedlock con un tiempo de espera de cero degenera en el caso trylock. En general, se deben evitar ambas versiones; sin embargo, hay algunos casos en los que puede ser útil evitar quedarse atascado (quizás indefinidamente) en una rutina de adquisición de bloqueo, como veremos en capítulos futuros (por ejemplo, cuando estudiemos el punto muerto).

## 27.4 Variables de condición

El otro componente importante de cualquier biblioteca de subprocessos, y ciertamente el caso de los subprocessos POSIX, es la presencia de una variable de condición. Las variables de condición son útiles cuando debe tener lugar algún tipo de señalización entre subprocessos, si un subprocesso está esperando que otro haga algo antes de poder continuar. Los programas que desean interactuar de esta manera utilizan dos rutinas principales:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex); int pthread_cond_signal(pthread_cond_t *cond);
```

Para utilizar una variable de condición, además hay que tener un candado asociado a esta condición. Al llamar a cualquiera de las rutinas anteriores, se debe mantener este bloqueo.

La primera rutina, pthread cond wait(), pone en suspensión el subprocesso que realiza la llamada y, por lo tanto, espera a que algún otro subprocesso lo indique, normalmente cuando algo en el programa ha cambiado y que podría interesarle al subprocesso que ahora está inactivo. Un uso típico se ve así:

```
pthread_mutex_t bloqueo = PTHREAD_MUTEX_INITIALIZER; pthread_cond_t
cond = PTHREAD_COND_INITIALIZER;
```

```
Pthread_mutex_lock(&bloqueo); mientras
(listo == 0)
    Pthread_cond_wait(&cond, &lock);
Pthread_mutex_unlock(&bloqueo);
```

En este código, después de la inicialización del bloqueo relevante y el hilo condition3 , a se verifica si la variable ready aún se ha establecido en un valor distinto de cero. De lo contrario, el hilo simplemente llama a la rutina de espera para dormir hasta que otro hilo lo despierte.

El código para activar un subprocesso, que se ejecutaría en algún otro subprocesso, tiene este aspecto:

```
Pthread_mutex_lock(&bloqueo); lista = 1;
```

```
Pthread_cond_signal(&cond);
Pthread_mutex_unlock(&bloqueo);
```

Algunas cosas a tener en cuenta sobre esta secuencia de código. Primero, al señalar (así como al modificar la variable global lista), siempre nos aseguramos de tener el bloqueo mantenido. Esto garantiza que no introduzcamos accidentalmente una condición de carrera en nuestro código.

En segundo lugar, puede notar que la llamada en espera toma un bloqueo como segundo parámetro, mientras que la llamada de señal solo toma una condición. La razón de esta diferencia es que la llamada en espera, además de poner a dormir el hilo de llamada, libera el bloqueo al poner a dormir a dicha persona que llama.

Imagínese si no fuera así: ¿cómo podría el otro hilo adquirir el bloqueo y indicarle que se despierte? Sin embargo, antes de regresar después de ser despertado, pthread cond wait() vuelve a adquirir el bloqueo, asegurando así que cada vez que el hilo en espera se ejecute entre la adquisición del bloqueo al comienzo de la secuencia de espera y la liberación del bloqueo al final, sostiene la cerradura.

Una última rareza: el hilo en espera vuelve a verificar la condición en un bucle while, en lugar de una simple declaración if. Discutiremos este tema en detalle cuando estudiemos las variables de condición en un capítulo futuro, pero en general, usar un bucle while es lo más simple y seguro. Aunque vuelve a verificar la condición (quizás agregando un poco de sobrecarga), hay algunas implementaciones de pthread que podrían activar falsamente un hilo en espera; en tal caso, sin volver a verificar, el hilo en espera seguirá pensando que la condición ha cambiado aunque no sea así. Por lo tanto, es más seguro ver el despertar como un indicio de que algo podría haber cambiado, en lugar de un hecho absoluto.

Tenga en cuenta que a veces resulta tentador utilizar una bandera simple para señalar entre dos subprocessos, en lugar de una variable de condición y un bloqueo asociado. Por ejemplo, podríamos reescribir el código de espera anterior para que se parezca más a esto en el código de espera:

```
mientras (listo == 0); // girar
```

El código de señalización asociado tendría este aspecto: lista = 1;

---

<sup>3</sup>Se puede utilizar pthread cond init() (y pthread cond destroy()) en lugar del Inicializador estático PTHREAD\_COND\_INITIALIZER. ¿Suena como más trabajo? Es.

Nunca hagas esto por las siguientes razones. En primer lugar, su rendimiento es deficiente en muchos casos (girar durante mucho tiempo sólo desperdicia ciclos de CPU). En segundo lugar, es propenso a errores. Como muestra una investigación reciente [X+10], es sorprendentemente fácil cometer errores al usar banderas (como arriba) para sincronizar entre subprocessos; En ese estudio, aproximadamente la mitad de los usos de estas sincronizaciones ad hoc tenían errores. No seas perezoso; utilice variables de condición incluso cuando crea que puede evitar hacerlo.

Si las variables de condición le parecen confusas, no se preocupe demasiado (todavía): las cubriremos con gran detalle en un capítulo posterior. Hasta entonces, debería bastar con saber que existen y tener una idea de cómo y por qué se utilizan.

## 27.5 Compilación y ejecución

Todos los ejemplos de código de este capítulo son relativamente fáciles de poner en marcha. Para compilarlos, debes incluir el encabezado `pthread.h` en tu código. En la línea de enlace, también debe vincular explícitamente con la biblioteca `pthreads`, agregando el indicador `-pthread`.

Por ejemplo, para compilar un programa multiproceso simple, todo lo que tienes que hacer es lo siguiente:

```
símbolo> gcc -o main main.c -Wall -pthread
```

Mientras `main.c` incluya el encabezado `pthreads`, habrá compilado exitosamente un programa concurrente. Si funciona o no, como siempre, es un asunto completamente diferente.

## 27.6 Resumen

Hemos introducido los conceptos básicos de la biblioteca `pthread`, incluida la creación de subprocessos, la creación de exclusión mutua mediante bloqueos y la señalización y espera mediante variables de condición. No necesitas mucho más para escribir código multiproceso robusto y eficiente, ¡excepto paciencia y mucho cuidado!

Ahora terminamos el capítulo con una serie de consejos que pueden resultarle útiles cuando escriba código multiproceso (consulte el apartado de la página siguiente para obtener más detalles). Hay otros aspectos de la API que son interesantes; Si desea obtener más información, escriba `man -k pthread` en un sistema Linux para ver más de cien API que conforman la interfaz completa. Sin embargo, los conceptos básicos discutidos aquí deberían permitirle crear programas multiproceso sofisticados (y, con suerte, correctos y eficaces). La parte difícil con los subprocessos no son las API, sino la lógica complicada de cómo crear programas concurrentes. Continúe leyendo para obtener más información.

**ADEMÁS: DIRECTRICES API DE HILO**

Hay una serie de cosas pequeñas pero importantes que debe recordar cuando utiliza la biblioteca de subprocesos POSIX (o en realidad, cualquier biblioteca de subprocesos) para crear un programa de subprocesos múltiples. Ellos son:

- Mantenerlo simple. Por encima de todo, cualquier código para bloquear o señalar entre subprocesos debe ser lo más simple posible. Las interacciones complicadas entre hilos generan errores.
- Minimizar las interacciones entre hilos. Intente mantener al mínimo el número de formas en que interactúan los subprocesos. Cada interacción debe ser cuidadosamente pensada y construida con enfoques probados y verdaderos (muchos de los cuales aprenderemos en los próximos capítulos).
- Inicializar bloqueos y variables de condición. No hacerlo conducirá a un código que a veces funciona y otras falla de maneras muy extrañas.
- Verifique sus códigos de retorno. Por supuesto, en cualquier programación en C y UNIX que realice, debe verificar todos y cada uno de los códigos de retorno, y esto también es cierto aquí. No hacerlo dará lugar a un comportamiento extraño y difícil de entender, lo que hará que sea probable que (a) grites, (b) te arranques un poco de pelo o (c) ambas cosas.
- Tenga cuidado con la forma en que pasa argumentos y devuelve valores de los subprocesos. En particular, cada vez que pasa una referencia a una variable asignada en la pila, probablemente esté haciendo algo mal.
- Cada hilo tiene su propia pila. En relación con el punto anterior, recuerde que cada hilo tiene su propia pila. Por lo tanto, si tiene una variable asignada localmente dentro de alguna función que está ejecutando un subproceso, es esencialmente privada para ese subproceso; ningún otro hilo puede acceder (fácilmente) a él. Para compartir datos entre subprocesos, los valores deben estar en el montón o en alguna ubicación que sea accesible globalmente.
- Utilice siempre variables de condición para señalar entre subprocesos. Si bien suele resultar tentador utilizar una bandera simple, no lo hagas.
- Utilice las páginas del manual. En Linux, en particular, las páginas de manual de pthread son muy informativas y analizan muchos de los matices presentados aquí, a menudo incluso con más detalle. ¡Léelos atentamente!

## Referencias

[B89] "Introducción a la programación con subprocessos" por Andrew D. Birrell. Informe técnico de DEC, enero de 1989. Disponible: <https://birrell.org/andrew/papers/035-Threads.pdf> Una introducción clásica pero más antigua a la programación con subprocessos. Sigue siendo una lectura que vale la pena y está disponible gratuitamente.

[B97] "Programación con subprocessos POSIX" de David R. Butenhof. Addison-Wesley, mayo de 1997.  
Otro de estos libros sobre hilos.

[B+96] "Programación de PThreads: según un estándar POSIX para un mejor multiprocesamiento." Dick Buttlar, Jacqueline Farrell, Bradford Nichols. O'Reilly, septiembre de 1996 Un libro razonable de la excelente y práctica editorial O'Reilly. Nuestras estanterías ciertamente contienen una gran cantidad de libros de esta empresa, incluidas algunas ofertas excelentes sobre Perl, Python y Javascript (en particular, "Javascript: The Good Parts" de Crockford).

[K+96] "Programación con subprocessos" de Steve Kleiman, Devang Shah, Bart Smaalders. Prentice Hall, enero de 1996. Probablemente uno de los mejores libros en este espacio. Consíguelo en tu biblioteca local. O robárselo a tu madre. Más en serio, pídeselo a tu madre; ella te lo prestará, no te preocupes.

[X+10] "Sincronización ad hoc considerada dañina" por Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, Zhiqiang Ma. OSDI 2010, Vancouver, Canadá. Este artículo muestra cómo un código de sincronización aparentemente simple puede generar una sorprendente cantidad de errores. ¡Utilice variables de condición y haga la señalización correctamente!

## Tarea (Código)

En esta sección, escribiremos algunos programas simples de subprocesos múltiples y usaremos una herramienta específica, llamada helgrind, para encontrar problemas en estos programas.

Lea el archivo README en la descarga de la tarea para obtener detalles sobre cómo crear los programas y ejecutar helgrind.

### Preguntas

1. Primero construye main-race.c. Examine el código para que pueda ver la carrera de datos (con suerte obvia) en el código. Ahora ejecuta helgrind (escribiendo valgrind --tool=helgrind main-race) para ver cómo informa la carrera. ¿Apunta a las líneas de código correctas? ¿Qué otra información te da?
2. ¿Qué sucede cuando eliminas una de las líneas de código ofensivas? Ahora agregue un bloqueo alrededor de una de las actualizaciones de la variable compartida y luego alrededor de ambas. ¿Qué informa helgrind en cada uno de estos casos?
3. Ahora veamos main-deadlock.c. Examina el código. Este código tiene un problema conocido como interbloqueo (que discutiremos con mucha más profundidad en un próximo capítulo). ¿Puedes ver qué problema puede tener?
4. Ahora ejecute helgrind en este código. ¿Qué informa Helgrind ?
5. Ahora ejecute helgrind en main-deadlock-global.c. Examinar el código; ¿Tiene el mismo problema que tiene main-deadlock.c ? ¿ Helgrind debería informar el mismo error? ¿Qué te dice esto sobre herramientas como helgrind?
6. Veamos a continuación main-signal.c. Este código utiliza una variable (hecho) para indicar que el niño ha terminado y que el padre ahora puede continuar. ¿Por qué este código es inefficiente? (¿En qué termina haciendo el padre su tiempo, especialmente si el hilo secundario tarda mucho en completarse?)
7. Ahora ejecute helgrind en este programa. ¿Qué informa? es el codigo correcto?
8. Ahora mire una versión ligeramente modificada del código, que se encuentra en main-signal-cv.c. Esta versión utiliza una variable de condición para realizar la señalización (y el bloqueo asociado). ¿Por qué se prefiere este código a la versión anterior? ¿Es corrección, desempeño o ambos?
9. Una vez más, ejecute helgrind en main-signal-cv. ¿Informa algún error?

## Cabellos

Desde la introducción a la concurrencia, vimos uno de los problemas fundamentales en la programación concurrente: nos gustaría ejecutar una serie de instrucciones de forma atómica, pero debido a la presencia de interrupciones en un solo procesador (o múltiples subprocesos ejecutándose en múltiples procesadores al mismo tiempo), no pudimos. En este capítulo, atacamos este problema directamente, con la introducción de algo llamado candado. Los programadores anotan el código fuente con bloqueos, colocándolos alrededor de secciones críticas y así garantizan que dicha sección crítica se ejecute como si fuera una única instrucción atómica.

## 28.1 Cerraduras: la idea básica

Como ejemplo, supongamos que nuestra sección crítica se ve así, la actualización canónica de una variable compartida:

```
saldo = saldo + 1;
```

Por supuesto, son posibles otras secciones críticas, como agregar un elemento a una lista vinculada u otras actualizaciones más complejas a estructuras compartidas, pero por ahora nos limitaremos a este ejemplo simple. Para usar un candado, agregamos código alrededor de la sección crítica como este:

```
1 exclusión mutua lock_t; // algún bloqueo 'mutex' asignado globalmente
2 ...
3 bloqueo (&mutex); 4
saldo = saldo + 1; 5 desbloquear(&mutex);
```

Un candado es solo una variable y, por lo tanto, para usar uno, debe declarar una variable de candado de algún tipo (como el mutex anterior). Esta variable de bloqueo (o simplemente "bloqueo" para abreviar) mantiene el estado del bloqueo en cualquier instante. Está disponible (o desbloqueado o libre) y, por lo tanto, ningún hilo retiene el bloqueo, o está adquirido (o bloqueado o retenido) y, por lo tanto, exactamente un hilo retiene el bloqueo y presumiblemente está en una sección crítica. También podríamos almacenar otra información en el tipo de datos, como qué hilo contiene el bloqueo o una cola.

para ordenar la adquisición de la cerradura, pero información como esa está oculta para el usuario de la cerradura.

La semántica de las rutinas lock() y unlock() es simple. Llamar a la rutina lock() intenta adquirir el bloqueo; si ningún otro hilo retiene el bloqueo (es decir, está libre), el hilo adquirirá el bloqueo y entrará en la sección crítica; A veces se dice que este hilo es el propietario del candado. Si otro subproceso llama a lock() en esa misma variable de bloqueo (mutex en este ejemplo), no regresará mientras otro subproceso mantenga el bloqueo; de esta manera, se evita que otros hilos entren en la sección crítica mientras el primer hilo que sostiene el candado esté allí.

Una vez que el propietario del candado llama a unlock(), el candado vuelve a estar disponible (gratis). Si ningún otro hilo está esperando el bloqueo (es decir, ningún otro hilo ha llamado a lock() y está atascado allí), el estado del bloqueo simplemente se cambia a libre. Si hay subprocessos en espera (atascados en el bloqueo()), uno de ellos (eventualmente) notará (o será informado de) este cambio en el estado del bloqueo, adquirirá el bloqueo e ingresará a la sección crítica.

Los bloqueos proporcionan a los programadores una cantidad mínima de control sobre la programación. En general, consideramos los subprocessos como entidades creadas por el programador pero programadas por el sistema operativo, de cualquier forma que el sistema operativo elija. Los bloqueos devuelven parte de ese control al programador; Al colocar un candado alrededor de una sección de código, el programador puede garantizar que no pueda haber más de un hilo activo dentro de ese código. Por lo tanto, los bloqueos ayudan a transformar el caos que supone la programación tradicional del sistema operativo en una actividad más controlada.

## 28.2 Bloqueos de rosca P

El nombre que usa la biblioteca POSIX para un bloqueo es mutex, ya que se usa para proporcionar exclusión mutua entre subprocessos, es decir, si un subproceso está en la sección crítica, excluye la entrada de los demás hasta que haya completado la sección. Por lo tanto, cuando vea el siguiente código de subprocessos POSIX, debe comprender que está haciendo lo mismo que el anterior (nuevamente usamos nuestros contenedores que verifican errores al bloquear y desbloquear):

```
1 bloqueo pthread_mutex_t = PTHREAD_MUTEX_INITIALIZER;
2
3 Pthread_mutex_lock(&bloqueo); // envoltorio; sale en caso de falla 4 saldo = saldo + 1;
5 Pthread_mutex_unlock(&bloquear);
```

También puede notar aquí que la versión POSIX pasa una variable para bloquear y desbloquear, ya que es posible que estemos usando diferentes bloqueos para proteger diferentes variables. Hacerlo puede aumentar la concurrencia: en lugar de utilizar un gran candado cada vez que se accede a una sección crítica (una estrategia de bloqueo de grano grueso ), a menudo se protegerán diferentes datos y estructuras de datos con diferentes candados, permitiendo así que haya más subprocessos en código bloqueado a la vez (un enfoque más detallado ).

### 28.3 Construyendo una cerradura

A estas alturas, deberías tener cierta comprensión de cómo funciona una cerradura, desde la perspectiva de un programador. Pero, ¿cómo deberíamos construir una cerradura? ¿Qué soporte de hardware se necesita? ¿Qué sistema operativo admite? Es este conjunto de preguntas el que abordamos en el resto de este capítulo.

#### EL CRUZ: CÓMO CONSTRUIR UNA CERRADURA

¿Cómo podemos construir una cerradura eficiente? Las cerraduras eficientes brindan exclusión mutua a bajo costo y también pueden lograr algunas otras propiedades que analizamos a continuación. ¿Qué soporte de hardware se necesita? ¿Qué sistema operativo admite?

Para construir una cerradura que funcione, necesitaremos la ayuda de nuestro viejo amigo, el hardware, así como de nuestro buen amigo, el sistema operativo. A lo largo de los años, se han añadido distintas primitivas de hardware a los conjuntos de instrucciones de diversas arquitecturas informáticas; Si bien no estudiaremos cómo se implementan estas instrucciones (ese, después de todo, es el tema de una clase de arquitectura de computadoras), estudiaremos cómo usarlas para construir una primitiva de exclusión mutua como una cerradura. También estudiaremos cómo interviene el sistema operativo para completar el cuadro y permitirnos construir una biblioteca de bloqueo sofisticada.

### 28.4 Evaluación de cerraduras

Antes de construir cualquier bloqueo, primero debemos comprender cuáles son nuestros objetivos y, por lo tanto, preguntarnos cómo evaluar la eficacia de la implementación de un bloqueo en particular. Para evaluar si una cerradura funciona (y funciona bien), debemos establecer algunos criterios básicos. La primera es si la cerradura hace su tarea básica, que es garantizar la exclusión mutua. Básicamente, ¿funciona el bloqueo e impide que varios subprocessos entren en una sección crítica?

El segundo es la justicia. ¿Tiene cada hilo que compite por el candado una oportunidad justa de adquirirlo una vez que esté libre? Otra forma de ver esto es examinando el caso más extremo: ¿algún hilo que compite por el candado muere de hambre mientras lo hace y, por lo tanto, nunca lo obtiene?

El criterio final es el rendimiento, específicamente los gastos generales de tiempo agregados. mediante el uso de la cerradura. Hay algunos casos diferentes que vale la pena considerar aquí. Uno es el caso de no contienda; cuando se ejecuta un solo hilo y agarra y libera el bloqueo, ¿cuál es la sobrecarga de hacerlo? Otro es el caso en el que varios subprocessos compiten por el bloqueo de una única CPU; En este caso, ¿existen problemas de rendimiento? Finalmente, ¿cómo funciona el bloqueo cuando hay múltiples CPU involucradas y subprocessos en cada una compitiendo por el bloqueo? Al comparar estos diferentes escenarios, podemos comprender mejor el impacto en el rendimiento del uso de varias técnicas de bloqueo, como se describe a continuación.

## 28.5 Control de interrupciones

Una de las primeras soluciones utilizadas para proporcionar exclusión mutua fue desactivar interrupciones para secciones críticas; Esta solución fue inventada para Sistemas de un solo procesador. El código se vería así:

```

1 bloqueo vacío() {
2     Desactivar interrupciones();
3 }
4 desbloqueo nulo() {
5     Habilitar interrupciones();
6 }
```

Supongamos que estamos ejecutando un sistema de un solo procesador. Al desactivar las interrupciones (usando algún tipo de instrucción de hardware especial) antes de ingresar a una sección crítica, nos aseguramos de que el código dentro de la sección crítica La sección no se interrumpirá y, por lo tanto, se ejecutará como si fuera atómica. Cuando terminamos, volvemos a habilitar las interrupciones (nuevamente, mediante una instrucción de hardware) y así el programa continúa como de costumbre.

El principal aspecto positivo de este enfoque es su simplicidad. Ciertamente no lo haces Tienes que rascar la cabeza demasiado para descubrir por qué funciona esto. Sin interrupción, un hilo puede estar seguro de que el código que ejecuta se ejecutará y que ningún otro hilo interferirá con ello.

Lamentablemente, los aspectos negativos son muchos. En primer lugar, este enfoque requiere permitir que cualquier hilo de llamada realice una operación privilegiada (convirtiendo interrupciones de encendido y apagado), y así confiar en que no se abusa de esta función. Como Ya lo sabes, cada vez que se nos pide que confiemos en un programa arbitrario, probablemente estemos en problemas. Aquí, el problema se manifiesta de numerosas maneras: un programa codicioso podría llamar a lock() al comienzo de su ejecución y así monopolizar el procesador; peor aún, un programa errante o malicioso podría llamar a lock() y entrar en un bucle sin fin. en esto En este último caso, el sistema operativo nunca recupera el control del sistema y sólo queda Un recurso: reiniciar el sistema. El uso de la desactivación de interrupciones como solución de sincronización de propósito general requiere demasiada confianza en las aplicaciones.

En segundo lugar, el enfoque no funciona en multiprocesadores. Si son múltiples Los subprocesos se ejecutan en diferentes CPU y cada uno intenta ingresar al mismo sección crítica, no importa si las interrupciones están deshabilitadas; trapos podrá ejecutarse en otros procesadores y, por tanto, podría entrar en la fase crítica sección. Como los multiprocesadores son ahora algo común, nuestra solución general Tendré que hacerlo mejor que esto.

En tercer lugar, desactivar las interrupciones durante períodos prolongados puede provocar las interrupciones se pierden, lo que puede provocar graves problemas en el sistema. Imagínese, por ejemplo, si la CPU pasara por alto el hecho de que un dispositivo de disco tiene finalizó una solicitud de lectura. ¿Cómo sabrá el sistema operativo que debe activar el proceso que espera dicha lectura?

Por último, y probablemente lo menos importante, este enfoque puede resultar ineficaz. En comparación con la ejecución normal de instrucciones, el código que enmascara o desenmascara Las interrupciones tienden a ejecutarse lentamente en las CPU modernas.

```

1 estructura typedef __lock_t { int bandera; } bloqueo_t;
2
3 inicio vacío (lock_t *mutex) {
4     // 0 -> el bloqueo está disponible, 1 -> retenido
5     mutex->bandera = 0;
6 }
7
8 bloqueo vacío (lock_t *mutex) {
9     while (mutex->flag == 1) // PRUEBA la bandera
10        ; // girar-esperar (no hacer nada)
11     mutex->bandera = 1;           // ahora ¡CONFIGÚRALO!
12 }
13
14 desbloqueo vacío (lock_t *mutex) {
15     mutex->bandera = 0;
16 }
```

Figura 28.1: Primer intento: una bandera simple

Por estas razones, desactivar las interrupciones sólo se utiliza en contextos limitados como una primitiva de exclusión mutua. Por ejemplo, en algunos casos un El propio sistema operativo utilizará el enmascaramiento de interrupciones para garantizar la atomicidad al acceder a sus propias estructuras de datos, o al menos para evitar que surjan ciertas situaciones desordenadas en el manejo de interrupciones. Este uso hace sentido, ya que el problema de confianza desaparece dentro del sistema operativo, que siempre confía sí mismo para realizar operaciones privilegiadas de todos modos.

## 28.6 Un intento fallido: simplemente usando cargas/almacenamiento

Para ir más allá de las técnicas basadas en interrupciones, tendremos que confiar en Hardware de la CPU y las instrucciones que nos proporciona para construir una cerradura adecuada. Primero intentemos construir un candado simple usando una única variable de bandera. En esto intento fallido, veremos algunas de las ideas básicas necesarias para construir una cerradura, y (con suerte) ver por qué simplemente usar una única variable y acceder a ella a través de Las cargas y almacenamientos normales son insuficientes.

En este primer intento (Figura 28.1), la idea es bastante simple: usar un simple variable (bandera) para indicar si algún hilo tiene posesión de un candado. El primer hilo que ingresa a la sección crítica llamará a lock(), que prueba si la bandera es igual a 1 (en este caso, no lo es) y luego establece la bandera a 1 para indicar que el hilo ahora mantiene el candado. Cuando termine con la sección crítica, el hilo llama a unlock() y borra la bandera, indicando así que el bloqueo ya no se mantiene.

Si otro hilo llama a lock() mientras ese primer hilo está en la sección crítica, simplemente girará y esperará en el ciclo while para esa hilo para llamar a unlock() y borrar la bandera. Una vez que ese primer hilo lo haga entonces, el hilo en espera saldrá del bucle while, establezca la bandera en 1 para y continúe con la sección crítica.

Desafortunadamente, el código tiene dos problemas: uno de corrección y otro.

Bloqueo de	Hilo 2
llamada del hilo 1 ()	
mientras (bandera == 1)	
interrupción: cambiar al hilo 2	
	bloqueo de llamada ()
	mientras (bandera == 1)
	bandera = 1;
	interrupción: cambiar al hilo 1
bandera = 1; // establece la bandera en 1 (¡también!)	

Figura 28.2: Seguimiento: sin exclusión mutua

otro de desempeño. El problema de corrección es fácil de ver una vez que Acostúmbrate a pensar en programación concurrente. imagina el código entrelazado en la Figura 28.2; asuma flag=0 para comenzar.

Como puede ver en este entrelazado, con interrupciones oportunas (¿intempestivas?), podemos producir fácilmente un caso en el que ambos subprocessos establezcan el indicador en 1. y ambos hilos pueden así entrar en la sección crítica. este comportamiento es lo que los profesionales llaman "malo": obviamente no hemos logrado brindar la requisito más básico: proporcionar exclusión mutua.

El problema de rendimiento, que abordaremos más adelante, es el hecho de que la forma en que un hilo espera adquirir un bloqueo que ya está retenido: comprueba sin cesar el valor de la bandera, una técnica conocida como espera de giro. La espera de giro es una pérdida de tiempo esperando a que otro hilo libere un bloqueo. El desperdicio es excepcionalmente alto en un monoprocesador, donde el hilo que ¡El camarero está esperando ni siquiera puede ejecutarse (al menos, hasta que ocurra un cambio de contexto)! Por lo tanto, a medida que avanzamos y desarrollamos soluciones más sofisticadas, también debemos considerar formas de evitar este tipo de desperdicio.

## 28.7 Construcción de cerraduras giratorias que funcionen con Test-And-Set

Debido a que deshabilitar las interrupciones no funciona en múltiples procesadores, y porque los enfoques simples que utilizan cargas y almacenajes (como se muestra arriba) no funcionan, los diseñadores de sistemas comenzaron a inventar soporte de hardware para el bloqueo. Los primeros sistemas multiprocesador, como el Burroughs B5000 en principios de los años 1960 [M82], contaba con ese apoyo; Hoy en día todos los sistemas proporcionan esto. tipo de soporte, incluso para sistemas con una sola CPU.

La parte de soporte de hardware más simple de entender se conoce como Instrucción de prueba y configuración (o intercambio atómico1). Definimos lo que hace la instrucción de prueba y configuración mediante el siguiente fragmento de código C:

```
1 int TestAndSet(int *antiguo_ptr, int nuevo) {
2     int antiguo = *old_ptr; // recupera el valor anterior en old_ptr
3     *old_ptr = nuevo; volver           // almacena 'nuevo' en old_ptr
4     viejo;                      // devuelve el valor anterior
5 }
```

1Cada arquitectura que admite prueba y configuración la llama con un nombre diferente. En SPARC es llamada instrucción de carga/almacenamiento de bytes sin firmar (ldstub); en x86 es la versión bloqueada del intercambio atómico (xchg).

### APARTE: LOS ALGORITMOS DE DEKKER Y PETERSON

En la década de 1960, Dijkstra planteó el problema de la concurrencia a sus amigos, y uno de ellos, un matemático llamado Theodorus Jozef Dekker, encontró una solución [D68]. A diferencia de las soluciones que discutimos aquí, que usan instrucciones de hardware especiales e incluso soporte para sistema operativo, el algoritmo de Dekker usa solo cargas y almacenes (asumiendo que son atómicos entre sí, lo cual era cierto en el hardware anterior).

El enfoque de Dekker fue posteriormente perfeccionado por Peterson [P81]. Una vez más, solo se utilizan cargas y almacenes, y la idea es garantizar que dos subprocesos nunca entren en una sección crítica al mismo tiempo. Aquí está el algoritmo de Peterson (para dos subprocesos); mira si puedes entender el código. ¿Para qué se utilizan las variables bandera y giro ?

bandera int[2]; a su vez;

```
inicio vacío() {
    // indica que tienes la intención de mantener el bloqueo con 'flag' flag[0] = flag[1] =
    0; // ¿a quién le toca? (hilo 0 o 1)
    turno = 0;

} void lock() { // 'self'
    es el ID del hilo del indicador de llamada [self] = 1; // hace que
    sea el turno de otro hilo
    turn = 1 - self; while ((flag[1-self] == 1) && (turno ==
    1 - self)); // gira-espera
    mientras no sea tu turno

} void unlock() {
    simplemente deshace tu indicador de
    intención[self] = 0;
}
```

Por alguna razón, el desarrollo de cerraduras que funcionan sin soporte de hardware especial se volvió furor por un tiempo, dando a los teóricos muchos problemas en los que trabajar. Por supuesto, esta línea de trabajo se volvió bastante inútil cuando la gente se dio cuenta de que era mucho más fácil asumir un poco de soporte de hardware (y de hecho, ese soporte había existido desde los primeros días del multiprocесamiento). Además, algoritmos como los anteriores no funcionan en hardware moderno (debido a modelos de consistencia de memoria relajados), lo que los hace aún menos útiles que antes. Aún hay más investigaciones relegadas al basurero de la historia...

```

1 estructura typedef __lock_t {
2         bandera interna;
3     } bloqueo_t;
4
5 inicio vacío (lock_t *bloqueo) {
6         // 0: el bloqueo está disponible, 1: el bloqueo está retenido
7         bloqueo->bandera = 0;
8     }
9
10 bloqueo vacío (lock_t *bloqueo) {
11         mientras (TestAndSet(&lock->flag, 1) == 1)
12             ; // girar-esperar (no hacer nada)
13     }
14
15 desbloqueo vacío (lock_t *lock) {
16         bloqueo->bandera = 0;
17     }

```

Figura 28.3: Un bloqueo de giro simple usando Test-and-set

Lo que hace la instrucción de prueba y configuración es lo siguiente. Devuelve lo viejo valor señalado por el antiguo ptr, y simultáneamente actualiza dicho valor a nuevo. La clave, por supuesto, es que esta secuencia de operaciones se realice atómicamente. La razón por la que se llama “probar y configurar” es que le permite para “probar” el valor anterior (que es lo que se devuelve) mientras simultáneamente “establecer” la ubicación de la memoria a un nuevo valor; resulta que esto es un poco una instrucción más poderosa es suficiente para construir un bloqueo de giro simple, como Ahora examinemos en la Figura 28.3. O mejor aún: ¡descubrela tú mismo primero!

Asegurémonos de entender por qué funciona este bloqueo. Imagínate primero el caso en el que un hilo llama a lock() y ningún otro hilo posee actualmente el cerrar con llave; por lo tanto, el indicador debe ser 0. Cuando el hilo llama a TestAndSet(flag, 1), la rutina devolverá el valor anterior de la bandera, que es 0; por lo tanto, el hilo que llama, que está probando el valor de la bandera, no quedará atrapado girando en el bucle while y adquirirá el bloqueo. El hilo también establecerá atómicamente el valor en 1, indicando así que el bloqueo ahora está mantenido. Cuando el hilo ha terminado con su sección crítica, llama a unlock() para establecer el bandera de nuevo a cero.

El segundo caso que podemos imaginar surge cuando un hilo ya tiene el bloqueo se mantiene (es decir, la bandera es 1). En este caso, este hilo llamará a lock() y luego llame también a TestAndSet(flag, 1) . Esta vez, TestAndSet() devolverá el valor anterior en la bandera, que es 1 (porque el bloqueo está mantenido), y al mismo tiempo lo configura nuevamente en 1. Mientras la cerradura esté sostenida por otro hilo, TestAndSet() devolverá repetidamente 1, y por lo tanto esto El hilo girará y girará hasta que finalmente se libere el bloqueo. Cuando la bandera está finalmente establecido en 0 por algún otro hilo, este hilo llamará a TestAndSet() nuevamente, que ahora devolverá 0 mientras establece atómicamente el valor en 1 y así adquirir la cerradura y entrar en la sección crítica.

Al realizar tanto la prueba (del antiguo valor de bloqueo) como el conjunto (del nuevo

**CONSEJO: PIENSE EN LA CONCURRENCIA COMO UN PLANIFICADOR MALICIOSO**

A partir de este ejemplo, puede tener una idea del enfoque que debe adoptar para comprender la ejecución concurrente. Lo que debería intentar hacer es fingir que es un programador malicioso, uno que interrumpe subprocessos en el momento más inoportuno para frustrar sus débiles intentos de construir primitivas de sincronización. ¡Qué programador tan malo eres! Aunque la secuencia exacta de interrupciones puede ser improbable, es posible, y eso es todo lo que necesitamos para demostrar que un enfoque particular no funciona.

¡Puede resultar útil pensar maliciosamente! (al menos, a veces)

valor) una sola operación atómica, nos aseguramos de que solo un subprocesso adquiera el bloqueo. ¡Y así es como se construye una primitiva de exclusión mutua que funcione!

Ahora también comprenderás por qué este tipo de bloqueo suele denominarse bloqueo giratorio. Es el tipo de candado más simple de construir y simplemente gira, usando ciclos de CPU, hasta que el candado esté disponible. Para funcionar correctamente en un solo procesador, se requiere un programador preventivo (es decir, uno que interrumpirá un subprocesso mediante un temporizador para ejecutar un subprocesso diferente, de vez en cuando). Sin preferencia, los bloqueos de giro no tienen mucho sentido en una sola CPU, ya que un hilo que gira en una CPU nunca lo abandonará.

## 28.8 Evaluación de bloqueos de giro

Dado nuestro bloqueo de giro básico, ahora podemos evaluar qué tan efectivo es a lo largo de nuestros ejes descritos anteriormente. El aspecto más importante de un candado es la corrección: ¿proporciona exclusión mutua? La respuesta aquí es sí: el bloqueo de giro solo permite que un hilo ingrese a la sección crítica a la vez. Así, tenemos un bloqueo correcto.

El siguiente eje es la equidad. ¿Qué tan justo es un bloqueo de giro para un hilo en espera? ¿Puede garantizar que un hilo en espera alguna vez ingrese a la sección crítica? Desafortunadamente, la respuesta aquí es una mala noticia: los bloqueos de giro no ofrecen ninguna garantía de equidad. De hecho, un hilo que gira puede girar para siempre, bajo disputa. Los bloqueos de giro simples (como se ha comentado hasta ahora) no son justos y pueden provocar inanición.

El último eje es el rendimiento. ¿Cuáles son los costos de usar un bloqueo giratorio? Para analizar esto más detenidamente, sugerimos pensar en algunos casos diferentes. En el primero, imagine subprocessos compitiendo por el bloqueo en un único procesador; en el segundo, considere los subprocessos distribuidos en muchas CPU.

Para los bloqueos de giro, en el caso de una sola CPU, los gastos generales de rendimiento pueden ser bastante dolorosos; Imagine el caso en el que el hilo que sujeta el candado se adelanta dentro de una sección crítica. El programador podría entonces ejecutar cada dos subprocessos (imáginate que hay  $N - 1$  más), cada uno de los cuales intenta adquirir el bloqueo. En este caso, cada uno de esos subprocessos girará durante un período de tiempo antes de abandonar la CPU, lo que supone una pérdida de ciclos de CPU.

Sin embargo, en varias CPU, los bloqueos de giro funcionan razonablemente bien (si la cantidad de subprocessos es aproximadamente igual a la cantidad de CPU). el pensamiento

```

1 int CompareAndSwap(int *ptr, int esperado, int nuevo) {
2     int original = *ptr;
3     si (original == esperado)
4         *ptr = nuevo;
5     devolver el original;
6 }
```

Figura 28.4: Comparar e intercambiar

dice lo siguiente: imagine el subprocesso A en la CPU 1 y el subprocesso B en la CPU 2, ambos compitiendo por un candado. Si el hilo A (CPU 1) agarra el candado y luego El subprocesso B lo intenta, B girará (en la CPU 2). Sin embargo, presumiblemente la sección crítica es corta y, por lo tanto, pronto el bloqueo estará disponible y el subprocesso B lo adquiere. Gira para esperar a que se mantenga un bloqueo en otro procesador. No desperdicia muchos ciclos en este caso y, por lo tanto, puede ser efectivo.

## 28.9 Comparar e intercambiar

Otra primitiva de hardware que proporcionan algunos sistemas se conoce como la instrucción comparar e intercambiar (como se llama en SPARC, por ejemplo), o comparar e intercambiar (como se llama en x86). El pseudocódigo C Para esta única instrucción se encuentra en la Figura 28.4.

La idea básica es comparar e intercambiar para probar si el valor en el la dirección especificada por ptr es igual a la esperada; si es así actualiza la memoria ubicación señalada por ptr con el nuevo valor. Si no, no hagas nada. En En cualquier caso, devuelve el valor original en esa ubicación de memoria, permitiendo así que el código que llama a comparar e intercambiar sepa si tuvo éxito o no.

Con la instrucción comparar e intercambiar, podemos construir un bloqueo de una manera bastante similar a la de probar y configurar. Por ejemplo, podríamos simplemente reemplace la rutina lock() anterior con lo siguiente:

```

1 bloqueo vacío (lock_t *bloqueo) {
2     mientras (CompareAndSwap(&lock->flag, 0, 1) == 1)
3         ; // girar
4 }
```

El resto del código es el mismo que el del ejemplo de prueba y configuración anterior. Este código funciona de manera bastante similar; simplemente comprueba si la bandera es 0 y si entonces, intercambia atómicamente un 1 adquiriendo así el bloqueo. Hilos que intentan adquirir el candado mientras está sostenido se quedará atascado girando hasta que el candado se finalmente liberado.

Si desea ver cómo hacer realmente una versión x86 invocable en C de comparar e intercambiar, la secuencia de códigos (de [S05]) puede ser útil<sup>2</sup>.

Finalmente, como habrás notado, comparar e intercambiar es una instrucción más poderosa que probar y configurar. Haremos algún uso de este poder en

---

<sup>2</sup>[github.com/remzi-arpacidusseau/ostep-code/tree/master/threads-locks](https://github.com/remzi-arpacidusseau/ostep-code/tree/master/threads-locks)

el futuro cuando profundizamos brevemente en temas como la sincronización sin bloqueo [H91]. Sin embargo, si construimos un simple bloqueo giratorio con él, es El comportamiento es idéntico al bloqueo de giro que analizamos anteriormente.

## 28.10 Vinculado a carga y condicional de almacenamiento

Algunas plataformas proporcionan un par de instrucciones que funcionan en conjunto para ayudar a construir secciones críticas. En la arquitectura MIPS [H93], por ejemplo, Las instrucciones vinculadas a la carga y condicionales de almacenamiento se pueden utilizar en conjunto. para construir cerraduras y otras estructuras concurrentes. El pseudocódigo C para estas instrucciones se encuentran en la Figura 28.5. Alfa, PowerPC y ARM proporciona instrucciones similares [W09].

La carga vinculada funciona de manera muy similar a una instrucción de carga típica, y simplemente recupera un valor de la memoria y lo coloca en un registro. La diferencia clave viene con el condicional de tienda, que sólo tiene éxito (y actualiza el valor almacenado en la dirección desde la que se acaba de cargar) si no hay intervención tienda a la dirección ha tenido lugar. En caso de éxito, el condicional de almacenamiento devuelve 1 y actualiza el valor en ptr a valor; si falla, el valor en ptr no se actualiza y se devuelve 0.

Como desafío para ti mismo, intenta pensar en cómo construir una cerradura usando vinculado a la carga y condicional al almacenamiento. Luego, cuando hayas terminado, mira el siguiente código que proporciona una solución simple. ¡Hazlo! la solución está en la Figura 28.6.

El código lock() es la única pieza interesante. Primero, un hilo gira esperando a que el indicador se establezca en 0 (y por lo tanto indique que el bloqueo no se mantiene). Una vez hecho esto, el hilo intenta adquirir el bloqueo a través del condicional de almacenamiento; si es tiene éxito, el hilo ha cambiado atómicamente el valor de la bandera a 1 y por lo tanto puede pasar a la sección crítica.

Tenga en cuenta cómo puede surgir una falla del condicional de almacenamiento. Un hilo llama lock() y ejecuta la carga vinculada, devolviendo 0 ya que el bloqueo no se mantiene. Antes de que pueda intentar la condición de almacenamiento, se interrumpe y otro El hilo ingresa el código de bloqueo y también ejecuta la instrucción vinculada a la carga.

```

1 int CargaEnlazada(int *ptr) {
2     devolver *ptr;
3 }
4
5 int StoreConditional(int *ptr, valor int) {
6     if (no hay actualización de *ptr desde LoadLinked a esta dirección) {
7         *ptr = valor;
8         devolver 1; // ¡éxito!
9     } demás {
10         devolver 0; // no se pudo actualizar
11     }
12 }
```

Figura 28.5: Vinculado a carga y condicional de almacenamiento

```

1 bloqueo vacío (lock_t *bloqueo) {
2     mientras (1) {
3         mientras (LoadLinked(&lock->flag) == 1)
4             ; // gira hasta que sea cero
5         si (StoreConditional(&lock->flag, 1) == 1)
6             devolver; // si configurarlo en 1 fue exitoso: todo listo
7             // en caso contrario: inténtalo todo de nuevo
8     }
9 }
10
11 desbloqueo vacío (lock_t *lock) {
12     bloqueo->bandera = 0;
13 }
```

Figura 28.6: Uso de LL/SC para construir una cerradura

y también obtener un 0 y continuar. En este punto, dos hilos tienen cada uno ejecutó el enlace de carga y cada uno está a punto de intentar el condicional de almacenamiento. La característica clave de estas instrucciones es que sólo uno de estos los subprocessos lograrán actualizar el indicador a 1 y así adquirir el bloqueo; el segundo hilo para intentar el condicional de almacenamiento fallará (porque el otro hilo actualizó el valor de la bandera entre su vínculo de carga y su condicional de almacenamiento) y, por lo tanto, tiene que intentar adquirir el bloqueo nuevamente.

En clase hace unos años, el estudiante universitario David Capel sugirió una forma más concisa de lo anterior, para aquellos de ustedes que disfrutan condicionales booleanos de cortocircuito. Vea si puede descubrir por qué es equivalente. ¡Ciertamente es más corto!

```

1 bloqueo vacío (lock_t *bloqueo) {
2     mientras (LoadLinked(&lock->flag) ||
3             !StoreConditional(&lock->bandera, 1))
4         ; // girar
5 }
```

## 28.11 Buscar y agregar

Una última primitiva de hardware es la instrucción de buscar y agregar , que incrementa átomicamente un valor mientras devuelve el valor anterior en una dirección particular. El pseudocódigo C para la instrucción buscar y agregar parece como esto:

```

1int FetchAndAdd(int *ptr) {
2     int antiguo = *ptr;
3     *ptr = antiguo + 1;
4     volver viejo;
5 }
```

**CONSEJO: MENOS CÓDIGO ES MEJOR CÓDIGO (LEY DE LAUER )**

Los programadores tienden a alardear de la cantidad de código que escribieron para hacer algo. Hacerlo es fundamentalmente roto. Más bien, de lo que uno debería presumir es del poco código que uno escribió para realizar una tarea determinada. Siempre se prefiere un código breve y conciso; probablemente sea más fácil de entender y tenga menos errores. Como dijo Hugh Lauer, al hablar de la construcción del sistema operativo Pilot: "Si las mismas personas tuvieran el doble de tiempo, podrían producir un sistema igual de bueno con la mitad del código". [L81] A esto lo llamaremos Ley de Lauer y vale la pena recordarlo. Así que la próxima vez que te jactes de la cantidad de código que escribiste para terminar la tarea, piénsalo de nuevo, o mejor aún, regresa, reescribe y haz que el código sea lo más claro y conciso posible.

En este ejemplo, usaremos buscar y agregar para crear un bloqueo de ticket más interesante, como lo presentaron Mellor-Crummey y Scott [MS91]. El código de bloqueo y desbloqueo se encuentra en la Figura 28.7 (página 14).

En lugar de un valor único, esta solución utiliza un ticket y una variable de giro en combinación para crear un candado. La operación básica es bastante simple: cuando un hilo desea adquirir un candado, primero realiza una búsqueda y adición atómica del valor del ticket; ese valor ahora se considera el "turno" de este hilo (myturn). El bloqueo->giro compartido globalmente se usa luego para determinar qué hilo es el turno; cuando (myturn == turn) para un hilo determinado, es el turno de ese hilo de ingresar a la sección crítica. El desbloqueo se logra simplemente incrementando el turno de modo que el siguiente hilo en espera (si lo hay) ahora pueda ingresar a la sección crítica.

Tenga en cuenta una diferencia importante entre esta solución y nuestros intentos anteriores: garantiza el progreso de todos los subprocessos. Una vez que a un hilo se le asigna su valor de ticket, se programará en algún momento en el futuro (una vez que los que están delante de él hayan pasado por la sección crítica y hayan liberado el bloqueo). En nuestros intentos anteriores, no existía tal garantía; un hilo que gira en prueba y configuración (por ejemplo) podría girar para siempre incluso cuando otros hilos adquieren y liberan el bloqueo.

## 28.12 Demasiados giros: ¿y ahora qué?

Nuestros bloqueos simples basados en hardware son simples (sólo unas pocas líneas de código) y funcionan (incluso podrías probarlo si lo deseas, escribiendo algún código), que son dos propiedades excelentes de cualquier sistema o código.

Sin embargo, en algunos casos, estas soluciones pueden resultar bastante ineficaces. Imagine que está ejecutando dos subprocessos en un solo procesador. Ahora imagine que un hilo (hilo 0) está en una sección crítica y, por lo tanto, tiene un bloqueo retenido y, lamentablemente, se interrumpe. El segundo hilo (hilo 1) ahora intenta adquirir el candado, pero descubre que está retenido. Así, comienza a girar. Y girar.

Luego gira un poco más. Y finalmente, se activa una interrupción del temporizador, el subprocesso 0 se ejecuta nuevamente, lo que libera el bloqueo y, finalmente (la próxima vez que se ejecute,

```

1 estructura typedef __lock_t {
2         billete internacional;
3         a su vez;
4     } bloqueo_t;
5
6 void lock_init(lock_t *bloqueo) {
7         bloqueo->boleto = 0;
8         bloquear->girar = 0;
9     }
10
11 bloqueo vacío (lock_t *bloqueo) {
12         int miturno = FetchAndAdd(&lock->ticket);
13         mientras (bloquear->girar! = migiro)
14             ; // girar
15         }
16
17 desbloqueo vacío (lock_t *lock) {
18         bloquear->girar = bloquear->girar + 1;
19     }

```

Figura 28.7: Bloqueos de boletos

digamos), el hilo 1 no tendrá que girar tanto y podrá adquirir el cerrar con llave. Por lo tanto, cada vez que un hilo queda atrapado girando en una situación como esta, desperdicia todo un intervalo de tiempo sin hacer nada más que comprobar un valor que no se va a cambiar! El problema empeora cuando N hilos compiten para una cerradura; N – 1 intervalos de tiempo se pueden desperdiciar de manera similar, simplemente girando y esperando que un solo hilo libere el bloqueo. Y así, nuestro siguiente problema:

#### EL CRUX: CÓMO EVITAR EL GIRO

¿Cómo podemos desarrollar un candado que no pierda tiempo innecesariamente girando la CPU?

El soporte de hardware por sí solo no puede resolver el problema. ¡También necesitaremos soporte para el sistema operativo! Ahora averigüemos cómo podría funcionar.

### 28.13 Un enfoque simple: simplemente cede, cariño

El soporte de hardware nos llevó bastante lejos: cerraduras que funcionan e incluso (como en el caso del candado de boleto) equidad en la adquisición del candado. Sin embargo, todavía Tengo un problema: qué hacer cuando se produce un cambio de contexto en un momento crítico. sección, y los hilos comienzan a girar sin cesar, esperando el interrumpido ¿Se ejecutará nuevamente el hilo (de retención de bloqueo)?

Nuestro primer intento es un enfoque simple y amigable: cuando vas a spin, en su lugar, ceda la CPU a otro hilo. Como diría Al Davis, "¡Solo ríndete, bebé!" [D91]. La figura 28.8 (página 15) muestra el enfoque.

```

1 inicio vacío() { bandera =
2         0;
3     }
4
5 bloqueo vacío() {
6     mientras (TestAndSet(&flag, 1) == 1) rendimiento(); //
7             renunciar a la CPU
8 }
9
10 desbloqueo vacío() { bandera
11     = 0;
12 }
```

Figura 28.8: Bloqueo con prueba y configuración y rendimiento

En este enfoque, asumimos un rendimiento primitivo() del sistema operativo al que un subprocesso puede llamar cuando quiere ceder la CPU y dejar que se ejecute otro subprocesso. Un hilo puede estar en uno de tres estados (en ejecución, listo o bloqueado); El rendimiento es simplemente una llamada al sistema que mueve a la persona que llama del estado de ejecución al estado listo y, por lo tanto, promueve la ejecución de otro subprocesso. Por lo tanto, el hilo elástico esencialmente se desprograma a sí mismo.

Piense en el ejemplo con dos subprocessos en una CPU; En este caso, nuestro enfoque basado en el rendimiento funciona bastante bien. Si un hilo llama a lock() y encuentra un bloqueo retenido, simplemente cederá la CPU y, por lo tanto, el otro hilo se ejecutará y finalizará su sección crítica. En este caso sencillo, el método de cesión funciona bien.

Consideremos ahora el caso en el que hay muchos subprocessos (digamos 100) compitiendo por un bloqueo repetidamente. En este caso, si un subprocesso adquiere el bloqueo y se adelanta antes de liberarlo, los otros 99 llamarán a lock(), encontrarán el bloqueo retenido y cederán la CPU. Suponiendo que haya algún tipo de programador por turnos, cada uno de los 99 ejecutarán este patrón de ejecución y rendimiento antes de que el subprocesso que mantiene el bloqueo se ejecute nuevamente. Si bien es mejor que nuestro enfoque de hilado (que desperdiciaría 99 porciones de tiempo girando), este enfoque sigue siendo costoso; El costo de un cambio de contexto puede ser sustancial y, por lo tanto, hay mucho desperdicio.

Peor aún, no hemos abordado en absoluto el problema del hambre. Un hilo puede quedar atrapado en un bucle de rendimiento sin fin mientras otros hilos entran y salen repetidamente de la sección crítica. Es evidente que necesitaremos un enfoque que aborde este problema directamente.

## 28.14 Uso de colas: dormir en lugar de girar

El verdadero problema con nuestros enfoques anteriores es que dejan demasiado al azar. El programador determina qué subprocesso se ejecuta a continuación; Si el programador toma una mala decisión, se ejecuta un subprocesso que debe girar esperando el bloqueo (nuestro primer enfoque) o ceder la CPU inmediatamente (nuestro segundo enfoque). De cualquier manera, existe la posibilidad de que se produzca despilfarro y no se puede prevenir la hambruna.

```

1 estructura typedef __lock_t {
2     bandera interna;
3     guardia interna;
4     cola_t *q;
5 } bloqueo_t;
6
7 anular lock_init(lock_t *m) {
8     m->bandera = 0;
9     m->guardia = 0;
10    cola_init(m->q);
11 }
12
13 bloqueo vacío (lock_t *m) {
14     mientras (TestAndSet(&m->guard, 1) == 1)
15         ; //adquirir bloqueo de guardia girando
16     si (m->bandera == 0) {
17         m->bandera = 1; // se adquiere el bloqueo
18         m->guardia = 0;
19     } demás {
20         queue_add(m->q, gettid());
21         m->guardia = 0;
22         parque();
23     }
24 }
25
26 desbloqueo vacío (lock_t *m) {
27     mientras (TestAndSet(&m->guard, 1) == 1)
28         ; //adquirir bloqueo de guardia girando
29     si (queue_empty(m->q))
30         m->bandera = 0; // suelta el bloqueo; nadie lo quiere
31     demás
32         desaparcar(queue_remove(m->q)); // mantener bloqueado
33                                         // (para el próximo hilo!)
34     m->guardia = 0;
35 }
```

Figura 28.9: Bloqueo con colas, prueba y configuración, rendimiento y activación

Por lo tanto, debemos ejercer explícitamente algún control sobre qué hilo sigue. adquiere el candado después de que el titular actual lo libera. Para hacer esto, nosotros Necesitará un poco más de compatibilidad con el sistema operativo, así como una cola para realizar un seguimiento de qué subprocesos están esperando adquirir el bloqueo.

Para simplificar, utilizaremos el soporte proporcionado por Solaris, en términos de dos llamadas: park() para poner un hilo de llamada en suspensión y unpark(threadID) para activar un hilo en particular según lo designado por threadID. Estas dos rutinas se pueden usar en conjunto para construir un bloqueo que duerma a la persona que llama si intenta adquirir un candado retenido y lo despierta cuando el candado está libre. miremos Consulte el código de la Figura 28.9 para comprender un posible uso de dichas primitivas.

**APARTE: MÁS RAZONES PARA EVITAR EL GIRO: INVERSIÓN DE PRIORIDAD Una buena razón**

para evitar los bloqueos de giro es el rendimiento: como se describe en el texto principal, si un subprocesso se interrumpe mientras mantiene un bloqueo, otros subprocessos que usan bloqueos de giro gastarán una gran cantidad de CPU tiempo simplemente esperando que el candado esté disponible. Sin embargo, resulta que hay otra razón interesante para evitar los bloqueos de giro en algunos sistemas: la corrección. ¡El problema del que hay que tener cuidado se conoce como inversión de prioridad, que desafortunadamente es un flagelo intergaláctico que ocurre en la Tierra [M15] y Marte [R97]!

Supongamos que hay dos subprocessos en un sistema. El subprocesso 2 (T2) tiene una prioridad de programación alta y el subprocesso 1 (T1) tiene una prioridad menor. En este ejemplo, supongamos que el programador de la CPU siempre ejecutará T2 sobre T1, si es que ambos son ejecutables; T1 sólo se ejecuta cuando T2 no puede hacerlo (por ejemplo, cuando T2 está bloqueado en E/S).

Ahora, el problema. Supongamos que T2 está bloqueado por algún motivo. Entonces T1 corre, toma un bloqueo de giro y entra en una sección crítica. T2 ahora se desbloquea (tal vez porque se completó una E/S) y el programador de la CPU lo programa inmediatamente (desprogramando así T1). T2 ahora intenta adquirir el candado y, como no puede (T1 retiene el candado), sigue girando. Debido a que la cerradura es una cerradura giratoria, T2 gira para siempre y el sistema se cuelga.

Desafortunadamente, simplemente evitar el uso de bloqueos de giro no evita el problema de la inversión (por desgracia). Imagine tres subprocessos, T1, T2 y T3, con T3 con la prioridad más alta y T1 con la más baja. Imaginemos ahora que T1 agarra un candado. Luego se inicia T3 y, debido a que tiene mayor prioridad que T1, se ejecuta inmediatamente (anticipando a T1). T3 intenta adquirir el candado que tiene T1, pero se queda atascado esperando porque T1 todavía lo tiene. Si T2 comienza a ejecutarse, tendrá mayor prioridad que T1 y, por lo tanto, se ejecutará. T3, que tiene mayor prioridad que T2, está atascado esperando a T1, que tal vez nunca se ejecute ahora que T2 está en ejecución. ¿No es triste que el poderoso T3 no pueda funcionar, mientras que el humilde T2 controla la CPU? Tener alta prioridad ya no es lo que solía ser.

Puede abordar el problema de la inversión de prioridad de varias maneras. En el caso específico en el que los bloqueos de giro causan el problema, puede evitar el uso de bloqueos de giro (que se describen con más detalle a continuación). De manera más general, un subprocesso de mayor prioridad que espera a un subprocesso de menor prioridad puede aumentar temporalmente la prioridad del subprocesso inferior, permitiéndole ejecutarse y superar la versión in, una técnica conocida como herencia de prioridad. Una última solución es la más sencilla: asegurarse de que todos los subprocessos tengan la misma prioridad.

Hacemos un par de cosas interesantes en este ejemplo. Primero, combinamos la antigua idea de probar y configurar con una cola explícita de camareros de cerraduras para hacer una cerradura más eficiente. En segundo lugar, utilizamos una cola para ayudar a controlar quién obtiene el siguiente candado y así evitar morir de hambre.

Podrías notar cómo se usa la guardia (Figura 28.9, página 16), básicamente como un bloqueo giratorio alrededor de la bandera y las manipulaciones de cola que usa el bloqueo. Por lo tanto, este enfoque no evita por completo la espera de giro; un hilo

podría interrumpirse al adquirir o liberar el bloqueo y, por lo tanto, provocar que otros subprocesos giren; espere a que este se ejecute nuevamente. Sin embargo, el tiempo dedicado a girar es bastante limitado (solo unas pocas instrucciones dentro del código de bloqueo y desbloqueo, en lugar de la sección crítica definida por el usuario) y, por lo tanto, este enfoque puede ser razonable.

También puede observar que en lock(), cuando un subproceso no puede adquirir el bloqueo (ya está retenido), tenemos cuidado de agregarlos a una cola (llamando a la función gettid() para obtener el ID del subproceso actual subproceso), establezca la protección en 0 y ceda la CPU. Una pregunta para el lector: ¿Qué pasaría si la liberación del bloqueo de guardia se produjera después del park(), y no antes? Pista: algo malo.

Es posible que detecte además que el indicador no vuelve a establecerse en 0 cuando se activa otro subproceso. ¿Por qué es esto? Bueno, ¡no es un error, sino una necesidad! Cuando se despierta un hilo, será como si volviera de park(); sin embargo, no mantiene la guardia en ese punto del código y por lo tanto ni siquiera puede intentar establecer el indicador en 1. Por lo tanto, simplemente pasamos el bloqueo directamente desde el hilo que libera el bloqueo al siguiente hilo que lo adquiere; El indicador no está establecido en 0 en el medio.

Finalmente, es posible que observe la condición de carrera percibida en la solución, justo antes de la llamada a park(). Simplemente con el momento equivocado, un hilo estará a punto de estacionarse, asumiendo que debería dormir hasta que el bloqueo ya no se mantenga. Un cambio en ese momento a otro hilo (digamos, un hilo que sostiene el bloqueo) podría generar problemas, por ejemplo, si ese hilo luego libera el bloqueo. El siguiente estacionamiento por parte del primer subproceso dormiría para siempre (potencialmente), un problema que a veces se denomina carrera de despertar/espera.

Solaris resuelve este problema agregando una tercera llamada al sistema: setpark(). Al llamar a esta rutina, un hilo puede indicar que está a punto de estacionarse. Si luego se interrumpe y otro hilo llama a unpark antes de que se llame realmente a park, el park posterior regresa inmediatamente en lugar de dormir. La modificación del código, dentro de lock(), es bastante pequeña:

```

1      queue_add(m->q, gettid()); setparque(); //
2      nuevo código m->guard = 0;
3

```

Una solución diferente podría pasar la protección al núcleo. En ese caso, el kernel podría tomar precauciones para liberar atómicamente el bloqueo y sacar de la cola el subproceso en ejecución.

## 28.15 Sistema operativo diferente, soporte diferente

Hasta ahora hemos visto un tipo de soporte que un sistema operativo puede proporcionar para crear un bloqueo más eficiente en una biblioteca de subprocesos. Otros sistemas operativos brindan soporte similar; los detalles varían.

Por ejemplo, Linux proporciona un futex que es similar a la interfaz de Solaris pero proporciona más funcionalidad interna del kernel. En concreto, cada futex tiene asociada una ubicación de memoria física específica, así como una

```

1 vacío mutex_lock (int *mutex) {
2     intv;
3     /* El bit 31 estaba claro, obtuvimos el mutex (la ruta rápida) */
4     si (atomic_bit_test_set (mutex, 31) == 0)
5         devolver;
6     incremento_atómico (mutex);
7     mientras (1) {
8         si (atomic_bit_test_set (mutex, 31) == 0) {
9             atomic_decrement (mutex);
10            devolver;
11        }
12        /* Tenemos que esperar, primero asegurarnos del valor de futex.
13           que estamos monitoreando es verdaderamente negativo (bloqueado). */
14        v = *mutex;
15        si (v >= 0)
16            continuar;
17        futex_wait (mutex,v);
18    }
19 }
20
21 vacío mutex_unlock (int *mutex) {
22     /* Agregar 0x80000000 al contador da como resultado 0 si y
23       sólo si no hay otros hilos interesados */
24     si (atomic_add_zero (mutex, 0x80000000))
25         devolver;
26
27     /* Hay otros hilos esperando este mutex,
28       despierta a uno de ellos.*/
29     futex_wake (mutex);
30 }
```

Figura 28.10: Cerraduras Futex basadas en Linux

Cola en el kernel por futex. Las personas que llaman pueden utilizar llamadas futex (descritas a continuación) dormir y despertar según sea necesario.

En concreto, hay dos convocatorias disponibles. La llamada a futex espera(dirección, esperado) pone el hilo de llamada en suspensión, asumiendo el valor en la dirección es igual a lo esperado. Si no es igual, la llamada regresa inmediatamente. El La llamada a la rutina futex wake(address) despierta un hilo que está esperando en la cola.

El uso de estas llamadas en un mutex de Linux se muestra en

Figura 28.10 (página 19).

Este fragmento de código de lowlevellock.h en la biblioteca nptl (parte de la biblioteca gnu libc) [L09] es interesante por varias razones. En primer lugar, utiliza un entero único para rastrear tanto si el bloqueo se mantiene como si no (el bit alto del número entero) y el número de camareros en la cerradura (todos los demás bits). Por lo tanto, si el bloqueo es negativo, se mantiene (porque el bit alto está establecido y eso El bit determina el signo del número entero).

En segundo lugar, el fragmento de código muestra cómo optimizar para el caso común,

específicamente cuando no hay disputa por la cerradura; con un solo hilo adquiriendo y liberando un bloqueo, se realiza muy poco trabajo (la prueba y configuración del bit atómico para bloquear y una adición atómica para liberar el bloqueo).

Vea si puede descifrar el resto de este candado del “mundo real” para comprender cómo funciona. Hazlo y conviértete en un maestro del bloqueo de Linux, o al menos en 3. alguien que escuche cuando un libro te dice que hagas algo.

## 28.16 Cerraduras bifásicas

Una nota final: el enfoque de Linux tiene el sabor de un enfoque antiguo que se ha utilizado intermitentemente durante años, remontándose al menos a Dahm Locks a principios de los años 1960 [M82], y ahora se lo conoce como un enfoque de dos bloqueo de fase. Una cerradura de dos fases comprende que el giro puede resultar útil, especialmente si la cerradura está a punto de abrirse. Entonces, en la primera fase, la cerradura gira por un tiempo, con la esperanza de poder adquirir la cerradura.

Sin embargo, si el bloqueo no se adquiere durante la primera fase de giro, se ingresa a una segunda fase, donde la persona que llama se pone a dormir y solo se despierta cuando el bloqueo se libera más tarde. El bloqueo de Linux anterior es una forma de dicho bloqueo, pero sólo gira una vez; una generalización de esto podría girar en bucle durante un período de tiempo fijo antes de usar el soporte futex para dormir.

Los bloqueos de dos fases son otro ejemplo de enfoque híbrido , en el que la combinación de dos buenas ideas puede generar una mejor. Por supuesto, que esto suceda depende en gran medida de muchas cosas, incluido el entorno de hardware, la cantidad de subprocesos y otros detalles de la carga de trabajo. Como siempre, crear una única cerradura de uso general, válida para todos los casos de uso posibles, es todo un desafío.

## 28.17 Resumen

El enfoque anterior muestra cómo se construyen las cerraduras reales hoy en día: algo de soporte de hardware (en forma de una instrucción más potente) más algo de soporte de sistema operativo (por ejemplo, en forma de primitivas park() y unpark() en Solaris, o futex . en Linux). Por supuesto, los detalles difieren y el código exacto para realizar dicho bloqueo suele estar muy ajustado. Consulte las bases de códigos de Solaris o Linux si desea ver más detalles; son una lectura fascinante [L09, S09]. Véase también el excelente trabajo de David et al. para una comparación de estrategias de bloqueo en multiprocesadores modernos [D+13].

---

3¡Me gusta comprar una copia impresa de OSTEP! Aunque el libro está disponible de forma gratuita en línea, ¿no le encantaría tener una tapa dura para su escritorio? O, mejor aún, ¿diez copias para compartir con amigos y familiares? ¿Y tal vez una copia extra para lanzarla a un enemigo? (El libro es pesado y, por lo tanto, tirarlo es sorprendentemente efectivo)

## Referencias

- [D91] "Simplemente gana, cariño: Al Davis y sus asaltantes" de Glenn Dickey. Harcourt, 1991. El libro sobre Al Davis y su famosa cita. O, suponemos, el libro trata más sobre Al Davis y los Raiders, y no tanto sobre la cita. Para ser claros: no recomendamos este libro, solo necesitábamos una cita.
- [D+13] "Todo lo que siempre quisiste saber sobre la sincronización pero temías preguntar" por Tudor David, Rachid Guerraoui, Vasileios Trigonakis. SOSP '13, Nemacolin Wood-lands Resort, Pensilvania, noviembre de 2013. Un excelente artículo que compara muchas formas diferentes de construir cerraduras utilizando primitivos de hardware. Es genial ver cuántas ideas funcionan en hardware moderno.
- [D68] "Procesos secuenciales cooperativos" por Edsger W. Dijkstra. 1968. Disponible en línea aquí: <http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF>. Uno de los primeros artículos fundamentales. Analiza cómo Dijkstra planteó el problema de concurrencia original y la solución de Dekker.
- [H93] "Manual del usuario del microprocesador MIPS R4000" por Joe Heinrich. Prentice-Hall, junio de 1993. Disponible: [http://cag.csail.mit.edu/raw/documents/R4400\\_Libro\\_Uman\\_Ed2.pdf](http://cag.csail.mit.edu/raw/documents/R4400_Libro_Uman_Ed2.pdf). El antiguo manual de usuario de MIPS. Descárgalo mientras aún existe.
- [H91] "Sincronización sin esperas" de Maurice Herlihy. ACM TOPLAS, Volumen 13: 1, enero de 1991. Un artículo histórico que presenta un enfoque diferente para la construcción de estructuras de datos concurrentes. Debido a la complejidad involucrada, algunas de estas ideas han tardado en ganar aceptación en su implementación.
- [L81] "Observaciones sobre el desarrollo de un sistema operativo" por Hugh Lauer. SOSP '81, Pacific Grove, California, diciembre de 1981. Una retrospectiva de lectura obligada sobre el desarrollo de Pilot OS, uno de los primeros sistemas operativos para PC. Divertido y lleno de ideas.
- [L09] "glibc 2.9 (incluye implementación de pthreads de Linux)" de muchos autores. Disponible aquí: <http://ftp.gnu.org/gnu/glibc/>. En particular, eche un vistazo al subdirectorio nptl donde encontrará la mayor parte del soporte pthread en Linux actualmente.
- [M82] "La arquitectura del Burroughs B5000: ¿20 años después y todavía por delante de los tiempos?" por A. Mayer. 1982. Disponible: [www.ajwm.net/amayer/papers/B5000.html](http://www.ajwm.net/amayer/papers/B5000.html). "(RDLK) es una operación indivisible que lee y escribe en una ubicación de memoria". ¡RDLK está, por tanto, probado y configurado! Dave Dahm creó cerraduras giratorias ("Buzz Locks") y una cerradura de dos fases llamada "Dahm Locks".
- [M15] "OSSpinLock no es seguro" por J. McCall. [mjtcsai.com/blog/2015/12/16/osspinlock-is-unsafe](http://mjtcsai.com/blog/2015/12/16/osspinlock-is-unsafe). Llamar a OSSpinLock en una Mac no es seguro cuando se utilizan subprocesos de diferentes prioridades: ¡podrías girar para siempre! Así que tengan cuidado, fanáticos de Mac, incluso su poderoso sistema puede no ser perfecto...
- [MS91] "Algoritmos para sincronización escalable en multiprocesadores de memoria compartida" por John M. Mellor-Crummey y ML Scott. ACM TOCS, Volumen 9, Número 1, febrero de 1991. Una encuesta excelente y exhaustiva sobre diferentes algoritmos de bloqueo. Sin embargo, no se utiliza ningún soporte para sistemas operativos, solo instrucciones sofisticadas de hardware.
- [P81] "Mitos sobre el problema de la exclusión mutua" por GL Peterson. Information Processing Letters, 12(3), páginas 115–116, 1981. Aquí se presenta el algoritmo de Peterson.
- [R97] "¿Qué pasó realmente en Marte?" por Glenn E. Reeves. [research.microsoft.com/en-us/um/people/mbr/Mars\\_Pathfinder/Authoritative\\_Account.html](http://research.microsoft.com/en-us/um/people/mbr/Mars_Pathfinder/Authoritative_Account.html). Una descripción de la inversión de prioridad en Mars Pathfinder. La corrección del código concurrente es importante, ¡especialmente en el espacio!
- [S05] "Guía para migrar de Solaris a Linux en x86" por Ajay Sood, 29 de abril de 2005. Disponible: <http://www.ibm.com/developerworks/linux/library/l-solar/>.
- [S09] "Biblioteca de subprocessos OpenSolaris" de Sun. Código: [src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/lib/libc/port/threads/synch.c](http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/lib/libc/port/threads/synch.c). Bastante interesante, aunque quién sabe qué pasará ahora que Oracle es dueño de Sun. Gracias a Mike Swift por el consejo.
- [W09] "Load-Link, Store-Conditional" de muchos autores... [en.wikipedia.org/wiki/Load-Link/Store-Conditional](http://en.wikipedia.org/wiki/Load-Link/Store-Conditional). ¿Puedes creer que hicimos referencia a Wikipedia? Pero encontramos la información allí y nos pareció mal no hacerlo. Además, fue útil enumerar las instrucciones para las diferentes arquitecturas: ldl l/stl c y ldq l/stq c (Alpha), lwarx/stwcx (PowerPC), ll/sc (MIPS) y Idrex/strex ( BRAZO). En realidad, Wikipedia es bastante sorprendente, así que no seas tan duro, ¿de acuerdo?
- [WG00] "Manual de arquitectura SPARC: Versión 9" por D. Weaver, T. Germond. SPARC Internacional, 2000. <http://www.sparc.org/standards/SPARCV9.pdf>. Ver desarrolladores. sun.com/solaris/articles/atomic\_sparc/ para obtener más información sobre la energía atómica.

## Tarea (Simulación)

Este programa, x86.py, le permite ver cómo diferentes intercalaciones de subprocessos causan o evitan condiciones de carrera. Consulte el archivo README para obtener detalles sobre cómo funciona el programa y responda las preguntas a continuación.

### Preguntas

1. Examinar las banderas. Este código "implementa" el bloqueo con una sola memoria bandera. ¿Puedes entender la asamblea?
2. Cuando ejecutas con los valores predeterminados, ¿funciona flag.s? Utilice los indicadores -M y -R para rastrear variables y registros (y active -c para ver sus valores). ¿Puedes predecir qué valor terminará en la bandera?
3. Cambie el valor del registro %bx con el indicador -a (por ejemplo, -a bx=2,bx=2 si está ejecutando solo dos subprocessos). ¿Qué hace el código? ¿Cómo cambia tu respuesta a la pregunta anterior?
4. Establezca bx en un valor alto para cada subprocesso y luego use el indicador -i para generar diferentes frecuencias de interrupción; ¿Qué valores conducen a malos resultados? ¿Cuáles conducen a buenos resultados?
5. Ahora veamos el programa test-and-set.s. Primero, intente comprender el código, que utiliza la instrucción xchg para construir una primitiva de bloqueo simple. ¿Cómo se escribe la adquisición de candado? ¿Qué tal el desbloqueo?
6. Ahora ejecute el código, cambiando el valor del intervalo de interrupción (-i) nuevamente y asegurándose de realizar un bucle varias veces. ¿El código siempre funciona como se esperaba? ¿Esto conduce a veces a un uso inefficiente de la CPU? ¿Cómo podrías cuantificar eso?
7. Utilice el indicador -P para generar pruebas específicas del código de bloqueo. Por ejemplo, ejecute una programación que capture el bloqueo en el primer hilo, pero luego intente adquirirlo en el segundo. ¿Sucede lo correcto? ¿Qué más deberías probar?
8. Ahora veamos el código en peterson.s, que implementa el algoritmo de Peterson (mencionado en una barra lateral del texto). Estudie el código y vea si puede entenderlo.
9. Ahora ejecute el código con diferentes valores de -i. ¿Qué tipo de comportamiento diferente ves? Asegúrese de configurar los ID de los subprocessos correctamente (usando -a bx=0,bx=1 por ejemplo) como lo asume el código.
10. ¿Puedes controlar la programación (con el indicador -P) para "probar" que el código funciona? ¿Cuáles son los diferentes casos que debes mostrar? Piense en la exclusión mutua y la evitación de puntos muertos.
11. Ahora estudie el código para el bloqueo del boleto en ticket.s. ¿Coincide con el código del capítulo? Luego ejecute con las siguientes banderas: -a bx=1000,bx=1000 (haciendo que cada hilo pase por la sección crítica 1000 veces). Mira lo que sucede; ¿Los hilos pasan mucho tiempo girando esperando el bloqueo?
12. ¿Cómo se comporta el código a medida que agregas más hilos?
13. Ahora examine los rendimientos, en los que una instrucción de rendimiento permite que un subprocesso ceda el control de la CPU (de manera realista, esto sería una primitiva del sistema operativo, pero por simplicidad, asumimos que una instrucción realiza la tarea). Encuentre un escenario en el que test-and-set.s desperdicia ciclos girando, pero produce.s no. ¿Cuántas instrucciones se guardan? ¿En qué escenarios surgen estos ahorros?
14. Finalmente, examine prueba y prueba y conjuntos. ¿Qué hace esta cerradura? ¿Qué tipo de ahorro introduce en comparación con los test-and-set.s?

## Variables de condición

Hasta ahora hemos desarrollado la noción de cerradura y hemos visto cómo se puede construir correctamente con la combinación adecuada de soporte de hardware y sistema operativo. Desafortunadamente, las cerraduras no son las únicas primitivas que se necesitan para construir programas concurrentes.

En particular, hay muchos casos en los que un hilo desea comprobar si una condición es verdadera antes de continuar su ejecución. Por ejemplo, Es posible que un hilo principal desee comprobar si un hilo secundario se ha completado antes de continuar (a esto se le suele llamar `join()`); ¿Cómo debería tal espera? implementarse? Miremos la Figura 30.1.

```

1 vacío *niño(vacio *arg) {
2     printf("niño\n");
3     // XXX ¿cómo indicar que hemos terminado?
4     devolver NULO;
5 }
6
7 int principal(int argc, char *argv[]) {
8     printf("padre: comenzar\n");
9     pthread_t c;
10    Pthread_create(&c, NULL, hijo, NULL); // crear niño
11    // XXX ¿cómo esperar al niño?
12    printf("padre: fin\n");
13    devolver 0;
14 }
```

Figura 30.1: Un padre esperando a su hijo

Lo que nos gustaría ver aquí es el siguiente resultado:

```

padre: comenzar
niño
padre: fin
```

Podríamos intentar usar una variable compartida, como se ve en la Figura 30.2. Este La solución generalmente funcionará, pero es enormemente ineficiente ya que el padre gira

```

1 int volátil hecho = 0;
2
3 vacío *niño(vacio *arg) {
4     printf("niño\n");
5     hecho = 1;
6     devolver NULO;
7 }
8
9 int principal(int argc, char *argv[]) {
10    printf("padre: comenzar\n");
11    pthread_tc;
12    Pthread_create(&c, NULL, hijo, NULL); // crear niño
13    mientras (hecho == 0)
14        ; // girar
15    printf("padre: fin\n");
16    devolver 0;
17 }
```

Figura 30.2: Padre esperando a su hijo: enfoque basado en giros

y desperdicia tiempo de CPU. En cambio, lo que nos gustaría aquí es alguna forma de poner a los padres a dormir hasta que se produzca la condición que estamos esperando (por ejemplo, el niño termina de ejecutar) se hace realidad.

#### EL CRUX : CÓMO ESPERAR UNA CONDICIÓN

En programas multiproceso, suele ser útil que un hilo espere alguna condición debe hacerse realidad antes de proceder. El enfoque simple, de simplemente girar hasta que la condición se vuelva verdadera, es tremadamente inefficiente y desperdicia ciclos de CPU y, en algunos casos, puede ser incorrecto. Así, ¿cómo ¿Debería un hilo esperar una condición?

### 30.1 Definición y rutinas

Para esperar a que una condición se cumpla, un hilo puede hacer uso de lo que se conoce como variable de condición. Una variable de condición es explícita. Cola en la que los subprocessos pueden ponerse cuando se encuentra en algún estado de ejecución. (es decir, alguna condición) no es la deseada (esperando la condición); algún otro hilo, cuando cambia dicho estado, puede despertar uno (o más) de esos hilos en espera y así permitirles continuar (al señalar la condición). La idea se remonta al uso que hace Dijkstra de "privado semáforos" [D68]; Una idea similar fue posteriormente denominada "variable de condición". por Hoare en su trabajo sobre monitores [H74].

Para declarar dicha variable de condición, simplemente se escribe algo así: pthread cond tc;, que declara c como variable de condición (nota: también se requiere una inicialización adecuada). Una variable de condición tiene dos operaciones asociadas con él: esperar () y señal (). La llamada de espera () se ejecuta cuando un hilo desea ponerse a dormir; la llamada de señal ()

```

1 int hecho = 0;
2 pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3 pthread_cond_t c = PTHREAD_COND_INITIALIZER;
4
5 anular thr_exit() {
6     Pthread_mutex_lock(&m);
7     hecho = 1;
8     Pthread_cond_signal(&c);
9     Pthread_mutex_unlock(&m);
10 }
11
12 vacío *niño(vacio *arg) {
13     printf("niño\n");
14     thr_exit();
15     devolver NULO;
16 }
17
18 vacío thr_join() {
19     Pthread_mutex_lock(&m);
20     mientras (hecho == 0)
21         Pthread_cond_wait(&c, &m);
22     Pthread_mutex_unlock(&m);
23 }
24
25 int principal(int argc, char *argv[]) {
26     printf("padre: comenzar\n");
27     pthread_tp;
28     Pthread_create(&p, NULL, hijo, NULL);
29     thr_join();
30     printf("padre: fin\n");
31     devolver 0;
32 }
```

Figura 30.3: Padre esperando a su hijo: utilice una variable de condición

se ejecuta cuando un hilo ha cambiado algo en el programa y por lo tanto quiere despertar un hilo dormido que espera esta condición. Específicamente, las llamadas POSIX se ven así:

```

pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
pthread_cond_signal(pthread_cond_t *c);
```

A menudo nos referiremos a ellos como esperar() y señal() por simplicidad. Una cosa que puede notar acerca de la llamada wait() es que también toma un mutex como parámetro; se supone que este mutex está bloqueado cuando se espera () se llama. La responsabilidad de esperar() es liberar el bloqueo y poner el llamar al hilo para que duerma (atómicamente); cuando el hilo se despierta (después de algunos otro hilo lo ha señalado), debe volver a adquirir el bloqueo antes de regresar a la persona que llama. Esta complejidad surge del deseo de prevenir ciertas

condiciones de carrera se produzcan cuando un hilo está tratando de ponerse a sí mismo dormir. Echemos un vistazo a la solución al problema de unión (Figura 30.3) para entender esto mejor.

Hay dos casos a considerar. En el primero, el padre crea al niño. hilo pero continúa ejecutándose (supongamos que tenemos un solo procesador) y por lo tanto llama inmediatamente a `thr join()` para esperar al hijo hilo para completar. En este caso adquirirá el candado, compruebe si el niño está hecho (no lo está) y se pone en modo de suspensión llamando a `wait()` (liberando así el bloqueo). El niño eventualmente correrá, imprimirá el mensaje "niño", y llame a `thr exit()` para activar el hilo principal; este código simplemente toma el lock, establece la variable de estado como terminada y le indica al padre que la active. Finalmente, el padre ejecutará (regresando de `wait()` con el bloqueo mantenido), desbloquee la cerradura e imprima el mensaje final "padre: fin".

En el segundo caso, el niño se ejecuta inmediatamente después de la creación, establece hecho a 1, llama a la señal para despertar un hilo dormido (pero no hay ninguno, por lo que simplemente regresa) y listo. El padre luego ejecuta, llama a `thr join()`, ve – lo hecho es 1 y, por lo tanto, no espera y regresa.

Una última nota: es posible que observe que el padre usa un bucle `while` en su lugar. de solo una declaración `if` al decidir si esperar o no la condición.

Si bien esto no parece estrictamente necesario según la lógica del programa, Siempre es una buena idea, como veremos a continuación.

Para asegurarse de comprender la importancia de cada pieza del `thr exit()` y `thr join()` código, probemos algunas implementaciones alternativas. Primero, quizás se pregunte si necesitamos que se complete la variable de estado. ¿Qué pasaría si el código se pareciera al ejemplo siguiente? (Figura 30.4)

Lamentablemente, este enfoque no funciona. Imaginemos el caso donde el niño corre inmediatamente y llama a `thr exit()`-inmediatamente; en este caso, el niño hará una señal, pero no hay ningún hilo dormido en la condición. Cuando el padre se ejecuta, simplemente llamará a esperar y se quedará atascado; ningún hilo lo hará jamás despiértalo. A partir de este ejemplo, deberías apreciar la importancia de la variable de estado hecha; registra el valor que interesa a los hilos conocimiento. El sueño, la vigilia y el encierro se construyen en torno a él.

```

1 anular thr_exit() {
2     Pthread_mutex_lock(&m);
3     Pthread_cond_signal(&c);
4     Pthread_mutex_unlock(&m);
5 }
6
7 vacío thr_join() {
8     Pthread_mutex_lock(&m);
9     Pthread_cond_wait(&c, &m);
10    Pthread_mutex_unlock(&m);
11 }
```

Figura 30.4: Padres en espera: Sin variable de estado

```

1 void thr_exit() { hecho = 1;
2
3     Pthread_cond_signal(&c);
4 }
5
6 void thr_join() {si (hecho == 0)
7
8     Pthread_cond_wait(&c);
9 }
```

Figura 30.5: Padres en espera: Sin bloqueo

Aquí (Figura 30.5) hay otra implementación deficiente. En este ejemplo, imaginamos que no es necesario mantener presionado un candado para señalar y esperar. ¿Qué problema podría ocurrir aquí? ¡Piénsalo1 !

El problema aquí es una condición racial sutil. Específicamente, si el padre llama a `thr join()` y luego verifica el valor de `hecho`, verá que es 0 y, por lo tanto, intentará irse a dormir. Pero justo antes de que llame a esperar para ir a dormir, el padre es interrumpido y el niño corre. El niño cambia la variable de estado `hecho` a 1 y señala, pero no hay ningún subproceso esperando y, por lo tanto, no se activa ningún subproceso. Cuando el padre vuelve a correr, duerme para siempre, lo cual es triste.

Con suerte, a partir de este sencillo ejemplo de unión, podrá ver algunos de los requisitos básicos para utilizar las variables de condición correctamente. Para asegurarnos de que lo comprende, veamos ahora un ejemplo más complicado: el problema del productor/consumidor o del buffer acotado.

#### CONSEJO: SIEMPRE MANTENGA LA CERRADURA MIENTRAS SEÑALA

Aunque no es estrictamente necesario en todos los casos, probablemente sea más sencillo y mejor mantener el bloqueo mientras se indica cuando se utilizan variables de condición. El ejemplo anterior muestra un caso en el que debe mantener presionado el candado para que esté correcto; sin embargo, hay otros casos en los que probablemente esté bien no hacerlo, pero probablemente sea algo que deba evitar. Por lo tanto, para simplificar, mantenga presionado el candado cuando llame a la señal.

Lo contrario de este consejo, es decir, mantener el bloqueo al llamar a esperar, no es sólo un consejo, sino que está ordenado por la semántica de esperar, porque esperar siempre (a) supone que el bloqueo se mantiene cuando lo llamas, (b) libera dicho bloqueo cuando pone a dormir a la persona que llama, y (c) vuelve a adquirir el bloqueo justo antes de regresar. Por tanto, la generalización de este consejo es correcta: mantén el candado cuando llames a la señal o espera, y siempre estarás en buena forma.

---

<sup>1</sup>Tenga en cuenta que este ejemplo no es un código "real", porque la llamada a `pthread cond wait()` siempre requiere un mutex además de una variable de condición; Aquí, simplemente pretendemos que la interfaz no lo hace por el bien del ejemplo negativo.

```

1 búfer int; 2 enteros
= 0; // inicialmente, vacío
3
4 void put(valor int) { afirmar(cuenta ==
5     0); contar = 1; búfer = valor;
6
7
8     }
9
10 int get()
11     { afirmar(cuenta == 1); contar =
12         0; búfer de
13         retorno;
14     }

```

Figura 30.6: Rutinas Put y Get (v1)

## 30.2 El problema del productor/consumidor (búfer acotado)

El siguiente problema de sincronización que enfrentaremos en este capítulo se conoce como el problema del productor/consumidor, o a veces como el problema del buffer acotado, que fue planteado por primera vez por Dijkstra [D72]. De hecho, fue este mismo problema productor/consumidor el que llevó a Dijkstra y sus compañeros de trabajo a inventar el semáforo generalizado (que puede usarse como bloqueo o como variable de condición) [D01]; Aprenderemos más sobre los semáforos más adelante.

Imagine uno o más subprocessos productores y uno o más subprocessos consumidores. Los productores generan elementos de datos y los colocan en un búfer; los consumidores toman dichos elementos del buffer y los consumen de alguna manera.

Esta disposición ocurre en muchos sistemas reales. Por ejemplo, en un servidor web multiproceso, un productor coloca solicitudes HTTP en una cola de trabajo (es decir, el búfer limitado); Los hilos de consumo sacan solicitudes de esta cola y las procesan.

También se utiliza un buffer acotado cuando canalizas la salida de un programa a otro, por ejemplo, grep foo file.txt | baño -l. Este ejemplo ejecuta dos procesos simultáneamente; grep escribe líneas desde file.txt con la cadena foo en lo que cree que es la salida estándar; el shell UNIX redirige la salida a lo que se llama una tubería UNIX (creada por la llamada al sistema de tuberías). El otro extremo de este tubo está conectado a la entrada estándar del proceso wc, que simplemente cuenta el número de líneas en el flujo de entrada e imprime el resultado. Por tanto, el proceso grep es el productor; el proceso wc es el consumidor; entre ellos hay un búfer delimitado dentro del núcleo; usted, en este ejemplo, es sólo el usuario feliz.

Debido a que el buffer limitado es un recurso compartido, por supuesto debemos requerir acceso sincronizado a él, para que no surja una condición de carrera. Para comenzar a comprender mejor este problema, examinemos algo de código real.

Lo primero que necesitamos es un búfer compartido, en el que un productor coloca datos y del cual un consumidor los toma. Usemos solo un

---

<sup>2</sup>Aquí es donde te presentamos algo serio de inglés antiguo y la forma subjuntiva.

```

1 vacío *productor(vacio *arg) {
2     ent yo;
3     int bucles = (int) arg;
4     for (i = 0; i < bucles; i++) {
5         poner(yo);
6     }
7 }
8
9 vacío *consumidor(vacio *arg) {
10    mientras (1) {
11        int tmp = obtener();
12        printf("%d\n", tmp);
13    }
14 }
```

Figura 30.7: Hilos de Productor/Consumidor (v1)

entero para simplificar (ciertamente puedes imaginar colocar un puntero a una estructura de datos en esta ranura en su lugar), y las dos rutinas internas para poner un valor en el búfer compartido y obtener un valor del búfer. Ver Figura 30.6 (página 6) para más detalles.

Bastante simple, ¿no? La rutina put() supone que el buffer está vacío. (y verifica esto con una afirmación), y luego simplemente pone un valor en el búfer compartido y lo marca como lleno estableciendo el recuento en 1. La rutina get() hace lo contrario, configura el búfer como vacío (es decir, configura el recuento en 0) y devolver el valor. No se preocupe, este búfer compartido tiene solo una partida simple; Más adelante, lo generalizaremos a una cola que puede contener múltiples entradas, que serán aún más divertidas de lo que parece.

Ahora necesitamos escribir algunas rutinas que sepan cuándo está bien acceder el buffer para poner datos en él o sacarlos de él. las condiciones para esto debería ser obvio: solo coloque datos en el búfer cuando el recuento sea cero (es decir, cuando el búfer está vacío) y solo obtiene datos del búfer cuando el recuento es uno (es decir, cuando el búfer está lleno). Si escribimos la sincronización código tal que un productor coloca datos en un búfer lleno, o un consumidor obtiene datos de uno vacío, algo hemos hecho mal (y en este código, se activará una aserción).

Este trabajo se realizará mediante dos tipos de hilos, un conjunto de los cuales llamaremos a los subprocesos productores y al otro conjunto los llamaremos subprocesos consumidores. La figura 30.7 muestra el código de un productor que pone un entero en el búfer compartido se repite varias veces y un consumidor que saca los datos de ese buffer compartido (para siempre), cada vez que se imprime extrae el elemento de datos que extrajo del búfer compartido.

## Una solución rota

Ahora imaginemos que tenemos un solo productor y un solo consumidor. Obviamente las rutinas put() y get() tienen secciones críticas dentro ellos, ya que put() actualiza el búfer y get() lee de él. Sin embargo, poner un candado alrededor del código no funciona; necesitamos algo más.

```

1 bucle interno; // debe inicializarse en alguna parte...
2 cond_t cond;
3 mutex_t mutex;
4
5 vacío *productor(vacío *arg) {
6     ent yo;
7     for (i = 0; i < bucles; i++) {
8         Pthread_mutex_lock(&mutex); // p2
9         si (cuenta == 1) // p1
10            Pthread_cond_wait(&cond, &mutex); // p3
11            poner(yo); // p4
12            Pthread_cond_signal(&cond); // p5
13            Pthread_mutex_unlock(&mutex); // p6
14        }
15    }
16
17 vacío *consumidor(vacio *arg) {
18     ent yo;
19     for (i = 0; i < bucles; i++) {
20         Pthread_mutex_lock(&mutex); // c1
21         si (cuenta == 0) // c2
22            Pthread_cond_wait(&cond, &mutex); // c3
23            int tmp = obtener(); // c4
24            Pthread_cond_signal(&cond); // c5
25            Pthread_mutex_unlock(&mutex); // c6
26            printf("%d\n", tmp);
27        }
28    }

```

Figura 30.8: Productor/Consumidor: CV único y declaración If

No es sorprendente que algo más sean algunas variables de condición. en esto (roto) primer intento (Figura 30.8), tenemos una variable de condición única cond y mutex de bloqueo asociado.

Examinemos la lógica de señalización entre productores y consumidores.

Cuando un productor quiere llenar el buffer, espera a que esté vacío (p1-p3). El consumidor tiene exactamente la misma lógica, pero espera una respuesta diferente. condición: plenitud (c1-c3).

Con un solo productor y un solo consumidor, el código de la Figura 30.8 obras. Sin embargo, si tenemos más de uno de estos hilos (por ejemplo, dos consumidores), la solución tiene dos problemas críticos. ¿Cuáles son?

... (pausa aquí para pensar) ...

Entendamos el primer problema, que tiene que ver con la declaración if antes de la espera. Supongamos que hay dos consumidores (Tc1 y Tc2) y un productor (Tp). Primero, se ejecuta un consumidor (Tc1) ; adquiere el bloqueo (c1), comprueba si algún buffer está listo para el consumo (c2), y encuentra que ninguno lo es, espera (c3) (que libera el bloqueo).

Luego se ejecuta el productor (Tp) . Adquiere el bloqueo (p1), comprueba si todos

## VARIABLES DE CONDICIÓN

Estado Tc1	Estado Tc2	Estado Tp	Contar	Comentario
c1 Ejecutar c2	Listo	Listo 0		
Ejecutar c3	Listo	Listo 0		
Dormir	Listo	Listo 0	Dormir	Nada que conseguir
Dormir	Listo	p2 Ejecutar 0		Búfer ahora lleno
Suspender	Listo	p4 Ejecutar 1	Listo	Tc1 despertado
Ejecutar 1	Listo	Listo p6 Ejecutar 1		
				...
Listo	Listo	p1 Ejecutar 1		
Listo	Listo	p2 Ejecutar 1		
Listo	Listo	p3 Suspender 1	Búfer lleno; dormir	
Ready	c1 Run	Sleep 1	Tc2 se cuela Ready	
	c2 Run	Sleep 1		
Listo	c4 Ejecuta	Sleep 0... y tóma datos		
Listo	c5 Ejecutar	Listo 0	Tp despertado	
Listo	c6 Ejecutar	Listo 0		
c4 Correr	Listo	Listo 0	¡Oh, oh! Sin datos	

Figura 30.9: Seguimiento del hilo: solución rota (v1)

los buffers están llenos (p2), y al descubrir que ese no es el caso, continúa y llena el buffer (p4). El productor indica entonces que se ha almacenado un buffer lleno (p5). Fundamentalmente, esto hace que el primer consumidor (Tc1) deje de dormir en una variable de condición a la cola lista; Tc1 ahora puede ejecutarse (pero aún no está en funcionamiento). Luego el productor continúa hasta darse cuenta del buffer. está lleno, momento en el que duerme (p6, p1–p3).

Aquí es donde ocurre el problema: se cuela otro consumidor (Tc2) y consume el único valor existente en el búfer (c1, c2, c4, c5, c6, omitiendo la espera en c3 porque el búfer está lleno). Ahora supongamos que se ejecuta Tc1 ; justo antes de regresar de la espera, vuelve a adquirir el candado y luego regresa. Él luego llama a get() (c4), ¡pero no hay buffers para consumir! una afirmación disparadores y el código no ha funcionado como se deseaba. Claramente, deberíamos de alguna manera han impedido que Tc1 intentara consumir porque Tc2 se coló y consumió el único valor en el búfer que se había producido. La figura 30.9 muestra la acción que realiza cada subprocesso, así como su estado del programador.

(Listo, corriendo o durmiendo) a lo largo del tiempo.

El problema surge por una sencilla razón: después de que el productor despertó Tc1, pero antes de que Tc1 se ejecutara, el estado del búfer limitado cambió (gracias a Tc2). Señalar un hilo sólo los despierta; es por lo tanto un indicio de que el estado del mundo ha cambiado (en este caso, que se ha colocado un valor en el búfer), pero no hay garantía de que cuando se ejecute el subprocesso activado, el Estado seguirá siendo el deseado. Esta interpretación de lo que significa una señal a menudo se la conoce como semántica de Mesa, después de la primera investigación que construyó una variable de condición de tal manera [LR80]; el contraste, denominado

```

1 bucle interno;
2 cond_t cond;
3 mutex_t mutex;
4
5 vacío *productor(vacío *arg) {
6     ent yo;
7     for (i = 0; i < bucles; i++) {
8         Pthread_mutex_lock(&mutex); //p2
9         mientras (cuenta == 1) // p1
10            Pthread_cond_wait(&cond, &mutex); //p3
11            poner(yo); // p4
12            Pthread_cond_signal(&cond); // p5
13            Pthread_mutex_unlock(&mutex); // p6
14        }
15    }
16
17 vacío *consumidor(vacio *arg) {
18     ent yo;
19     for (i = 0; i < bucles; i++) {
20         Pthread_mutex_lock(&mutex); // c1
21         mientras (cuenta == 0) // c2
22            Pthread_cond_wait(&cond, &mutex); // c3
23            int tmp = obtener(); // c4
24            Pthread_cond_signal(&cond); // c5
25            Pthread_mutex_unlock(&mutex); // c6
26            printf("%d\n", tmp);
27        }
28    }

```

Figura 30.10: Productor/Consumidor: CV único y mientras

La semántica de Hoare es más difícil de construir pero proporciona una garantía más sólida. que el hilo activado se ejecutará inmediatamente después de ser activado [H74]. Prácticamente todos los sistemas jamás construidos emplean la semántica de Mesa.

### Mejor, pero aún roto: mientras, no si

Afortunadamente, esta solución es fácil (Figura 30.10): cambie el if por un tiempo. Piense por qué esto funciona; ahora el consumidor Tc1 se despierta y (con el bloqueo mantenido) vuelve a verificar inmediatamente el estado de la variable compartida (c2). Si el buffer está vacío en ese punto, el consumidor simplemente vuelve a dormir (c3). El corolario if también se cambia a un tiempo en el productor (p2).

Gracias a la semántica de Mesa, una regla sencilla para recordar con condición variables es utilizar siempre bucles while. A veces no es necesario volver a comprobar la condición, pero siempre es seguro hacerlo; simplemente hazlo y sé feliz.

Sin embargo, este código todavía tiene un error, el segundo de los dos problemas mencionados anteriormente. ¿Puedes verlo? Tiene algo que ver con el hecho de que solo hay una variable de condición. Intenta descubrir cuál es el problema. es, antes de seguir leyendo. ¡HAZLO! (pausa para pensar, o cierra los ojos...)

## VARIABLES DE CONDICIÓN

Tc1	Estado	Tc2	Estado	Tp	Estado	Contar	Comentario
c1 Ejecutar c2		Listo	Listo	0			
Ejecutar c3		Listo	Listo	0			
Dormir		Listo	Listo	0	c1 Ejecutar	0	Nada que conseguir
Dormir		c2 Ejecutar	¡listo 0				
Dormir		c3	Dormir	Listo 0	Dormir p1 Ejecutar 0		Nada que conseguir
Dormir							
Dormir		Dormir	p2 Ejecutar 0				Búfer ahora lleno
Dormir		Dormir	p4 Ejecutar 1	Dormir	p5		
Listo		Ejecutar 1	Dormir	p6 Ejecutar 1			Tc1 despertado
Listo							
Listo		Dormir	p1 Ejecutar 1				
Listo		Dormir	p2 Ejecutar 1				
Listo c2		Dormir	p3 Dormir 1	Dormir	1 Debe dormir (completo)		
Ejecutar c4		Dormir	Dormir 1	Vuelva a verificar la condición			
Ejecutar c5		Sleep	Sleep 0	Tc1 toma datos			
Ejecutar c6		Listo para dormir 0	¡Ups!	Desperté Tc2			
Ejecutar c1		Listo para dormir 0					
Ejecutar c2		Listo para dormir 0					
Ejecutar c3		Listo para dormir 0					
Dormir	0	Listo	Dormir 0	c2 Ejecutar	Dormir		Nada que conseguir
Dormir	c3	Figura		Dormir 0	Todos dormidos...		

## 30.11 del sueño: Seguimiento del hilo: Solución rota (v2)

Confirmemos que lo has descubierto correctamente, o tal vez confirmemos que

Ahora estás despierto y leyendo esta parte del libro. El problema ocurre cuando dos consumidores se ejecutan primero (Tc1 y Tc2) y ambos se quedan dormidos (c3).

Luego, el productor ejecuta, pone un valor en el búfer y activa uno de los consumidores (digamos Tc1). Luego, el productor retrocede (liberando y recuperando el bloqueo a lo largo del camino) e intenta poner más datos en el buffer; debido a que el buffer está lleno, el productor espera la condición

(por lo tanto durmiendo). Ahora, un consumidor está listo para ejecutarse (Tc1) y dos subprocesos están durmiendo bajo una condición (Tc2 y Tp). Estamos a punto de causar un problema: ¡Las cosas se están poniendo emocionantes!

El consumidor Tc1 luego se despierta regresando de esperar() (c3), vuelve a verificar la condición (c2), y al encontrar el buffer lleno, consume el valor (c4).

Este consumidor entonces, críticamente, señala la condición (c5), despertando sólo un hilo que está dormido. Sin embargo, ¿qué hilo debería activarse?

Debido a que el consumidor ha vaciado el buffer, claramente debería despertarse el productor. Sin embargo, si despierta al consumidor Tc2 (que definitivamente es posible, dependiendo de cómo se gestione la cola de espera), tenemos un problema. Específicamente, el consumidor Tc2 se despertará y encontrará el buffer vacío (c2), y volver a dormir (c3). El productor Tp, que tiene un valor

```

1 cond_t vacío, llenar;
2 mutex_t mutex;
3
4 vacío *productor(vacío *arg) {
5     ent yo;
6     for (i = 0; i < bucles; i++) {
7         Pthread_mutex_lock(&mutex);
8         mientras (cuenta == 1)
9             Pthread_cond_wait(&empty, &mutex);
10        poner(yo);
11        Pthread_cond_signal(&relleno);
12        Pthread_mutex_unlock(&mutex);
13    }
14}
15
16 vacío *consumidor(vacío *arg) {
17     ent yo;
18     for (i = 0; i < bucles; i++) {
19         Pthread_mutex_lock(&mutex);
20         mientras (cuenta == 0)
21             Pthread_cond_wait(&relleno, &mutex);
22         int tmp = obtener();
23         Pthread_cond_signal(&empty);
24         Pthread_mutex_unlock(&mutex);
25         printf("%d\n", tmp);
26    }
27}

```

Figura 30.12: Productor/Consumidor: dos CV y mientras

para poner en el buffer, se queda durmiendo. El otro hilo consumidor, Tc1, También vuelve a dormir. Los tres hilos quedan dormidos, un claro error; ver En la figura 30.11 se muestra el brutal paso a paso de esta terrible calamidad.

La señalización es claramente necesaria, pero debe ser más dirigida. un consumidor no debería despertar a otros consumidores, sólo a los productores, y viceversa.

## La solución única para productor/consumidor

La solución aquí es una vez más pequeña: use dos variables de condición, en lugar de uno, para indicar correctamente qué tipo de hilo debe despertarse cuando el estado del sistema cambie. La figura 30.12 muestra la código resultante.

En el código, los subprocessos productores esperan la condición de vacío y las señales se llenan. Por el contrario, los hilos de consumo esperan a que se llene y señalan que están vacíos. Por Al hacerlo, el segundo problema anterior se evita por diseño: un consumidor nunca puede despertar accidentalmente a un consumidor, y un productor nunca puede despertar accidentalmente a un productor.

```

1 búfer int[MAX];
2 int fill_ptr = 0;
3 int use_ptr = 0;
4 enteros = 0;
5
6 put nulo (valor int) {
7     buffer[fill_ptr] = valor;
8     fill_ptr = (fill_ptr + 1) % MAX;
9     contar++;
10 }
11
12 int obtener() {
13     int tmp = búfer[use_ptr];
14     use_ptr = (use_ptr + 1) % MAX;
15     contar--;
16     devolver tmp;
17 }
```

Figura 30.13: Las rutinas de compra y venta correctas

```

1 cond_t vacío, llenar;
2 mutex_t mutex;
3
4 vacío *productor(vacío *arg) {
5     ent yo;
6     for (i = 0; i < bucles; i++) {
7         Pthread_mutex_lock(&mutex); // p1
8         mientras (cuenta == MAX) // p2
9             Pthread_cond_wait(&empty, &mutex); // p3
10            poner(yo); // p4
11            Pthread_cond_signal(&relleno); // p5
12            Pthread_mutex_unlock(&mutex); // p6
13     }
14 }
```

```

16 vacío *consumidor(vacío *arg) {
17     ent yo;
18     for (i = 0; i < bucles; i++) {
19         Pthread_mutex_lock(&mutex); // c2
20         mientras (cuenta == 0)
21             Pthread_cond_wait(&relleno, &mutex); // c3
22         int tmp = obtener(); // c5 // c4
23         Pthread_cond_signal(&empty); // c6
24         Pthread_mutex_unlock(&mutex); printf("%d\n", tmp);
25
26     }
27 }
```

Figura 30.14: La correcta sincronización productor/consumidor

**CONSEJO: USE WHILE (NO IF) PARA CONDICIONES Al verificar**

una condición en un programa multiproceso, usar un bucle while siempre es correcto; usar solo una declaración if podría serlo, dependiendo de la semántica de la señalización. Por lo tanto, utilice siempre while y su código se comportará como se espera.

El uso de bucles while alrededor de comprobaciones condicionales también maneja el caso en el que se producen reactivaciones espurias. En algunos paquetes de subprocessos, debido a detalles de la implementación, es posible que dos subprocessos se activen aunque solo haya tenido lugar una señal [L11]. Las reactivaciones espurias son una razón más para volver a verificar la condición en la que espera un hilo.

**La solución correcta para el productor y el consumidor**

Ahora tenemos una solución funcional para productores/consumidores, aunque no completamente general. El último cambio que realizamos es permitir una mayor simultaneidad y eficiencia; Específicamente, agregamos más ranuras de búfer, de modo que se puedan producir múltiples valores antes de dormir y, de manera similar, se puedan consumir múltiples valores antes de dormir. Con un solo productor y consumidor, este enfoque es más eficiente ya que reduce los cambios de contexto; con múltiples productores o consumidores (o ambos), incluso permite que se produzca o consuma simultáneamente, aumentando así la concurrencia. Afortunadamente, es un pequeño cambio con respecto a nuestra solución actual.

El primer cambio para esta solución correcta está dentro de la propia estructura del buffer y los correspondientes put() y get() (Figura 30.13). También cambiamos ligeramente las condiciones que los productores y consumidores verifican para determinar si dormir o no. También mostramos la lógica correcta de espera y señalización (Figura 30.14). Un productor sólo duerme si todos los buffers están actualmente llenos (p2); de manera similar, un consumidor solo duerme si todos los buffers están actualmente vacíos (c2). Y así solucionamos el problema productor/consumidor; Es hora de sentarse y beber uno frío.

**30.3 Condiciones de cobertura**

Ahora veremos un ejemplo más de cómo se pueden utilizar las variables de condición. Este estudio de código está extraído del artículo de Lampson y Redell sobre Pilot [LR80], el mismo grupo que implementó por primera vez la semántica de Mesa descrita anteriormente (el lenguaje que usaron fue Mesa, de ahí el nombre).

El problema con el que se encontraron se muestra mejor a través de un ejemplo simple, en este caso en una biblioteca de asignación de memoria multiproceso simple. La Figura 30.15 muestra un fragmento de código que demuestra el problema.

Como puede ver en el código, cuando un subprocesso llama al código de asignación de memoria, es posible que tenga que esperar para liberar más memoria. Por el contrario, cuando un hilo libera memoria, indica que hay más memoria libre. Sin embargo, nuestro código anterior tiene un problema: ¿qué hilo en espera (puede haber más de uno) debe activarse?

```

1 // ¿cuántos bytes del montón están libres?
2 int bytesLeft = MAX_HEAP_SIZE;
3
4 // también necesita bloqueo y condición
5 cond_t c;
6 mutex_tm;
7
8 vacío *
9 asignar (tamaño int) {
10     Pthread_mutex_lock(&m);
11     mientras (bytesLeft < tamaño)
12         Pthread_cond_wait(&c, &m);
13     vacío *ptr = ...; // sacarme del montón
14     bytesLeft -= tamaño;
15     Pthread_mutex_unlock(&m);
16     devolver ptr;
17 }
18
19 vacío libre (vacío *ptr, tamaño int) {
20     Pthread_mutex_lock(&m);
21     bytesLeft += tamaño;
22     Pthread_cond_signal(&c); // ¿A quién señalar?
23     Pthread_mutex_unlock(&m);
24 }
```

Figura 30.15: Condiciones de cobertura: un ejemplo

Consideré el siguiente escenario. Supongamos que hay cero bytes libres; El hilo Ta llama a allocate(100), seguido del hilo Tb que solicita menos memoria llamando a allocate(10). Por lo tanto, tanto Ta como Tb esperan el acondicionarse yirse a dormir; tampoco hay suficientes bytes libres para satisfacer de estas solicitudes.

En ese punto, supongamos que un tercer hilo, Tc, llama a free(50). Desafortunadamente, cuando llama a una señal para activar un hilo en espera, es posible que no se active. el hilo de espera correcto, Tb, que está esperando que solo se transfieran 10 bytes. liberado; Ta debería seguir esperando, ya que todavía no hay suficiente memoria libre. De este modo, El código de la figura no funciona, ya que el hilo despierta otros hilos. no sabe qué hilo (o hilos) activar.

La solución sugerida por Lampson y Redell es sencilla: reemplace la llamada pthread cond signal() en el código anterior con una llamada a pthread cond broadcast(), que activa todos los hilos en espera. Por Al hacerlo, garantizamos que todos los subprocessos que deban activarse lo serán. El La desventaja, por supuesto, puede ser un impacto negativo en el rendimiento, ya que podríamos despertar innecesariamente muchos otros hilos en espera que no deberían (todavía) ser despierto. Esos hilos simplemente se despertarán, volverán a verificar la condición y luego vuelva inmediatamente a dormir.

Lampson y Redell llaman a esta condición condición de cobertura, ya que cubre todos los casos en los que un hilo necesita activarse (de forma conservadora); el costo, como hemos comentado, es que se podrían activar demasiados subprocessos.

El lector astuto también podría haber notado que podríamos haber usado este enfoque antes (ver el problema productor/consumidor con una sola variable de condición). Sin embargo, en ese caso teníamos a nuestra disposición una solución mejor y por eso la utilizamos. En general, si descubre que su programa sólo funciona cuando cambia sus señales a transmisiones (pero no cree que sea necesario), probablemente tenga un error; arreglarlo! Pero en casos como el asignador de memoria anterior, la transmisión puede ser la solución más sencilla disponible.

## 30.4 Resumen

Hemos visto la introducción de otra primitiva de sincronización importante más allá de los bloqueos: las variables de condición. Al permitir que los subprocesos entren en suspensión cuando el estado de algún programa no es el deseado, los CV nos permiten resolver claramente una serie de problemas de sincronización importantes, incluido el famoso (y aún importante) problema de productor/consumidor, así como las condiciones de cobertura. Aquí iría una frase final más dramática, como “Amaba al Gran Hermano” [O49].

## Referencias

[D68] "Procesos secuenciales cooperativos" por Edsger W. Dijkstra. 1968. Disponible en línea aquí: <http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF>. Otro clásico de Dijkstra; leer sus primeros trabajos sobre concurrencia le enseñará mucho de lo que necesita saber.

[D72] "Flujos de información que comparten un búfer finito" por EW Dijkstra. Information Processing Letters 1: 179–180, 1972. <http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD329.PDF> El famoso artículo que introdujo el problema productor/consumidor.

[D01] "Mis recuerdos del diseño de sistemas operativos" por EW Dijkstra. Abril de 2001. Disponible: <http://www.cs.utexas.edu/users/EWD/ewd13xx/EWD1303.PDF>. ¡Una lectura fascinante para aquellos de ustedes interesados en cómo los pioneros de nuestro campo idearon algunos conceptos muy básicos y fundamentales, incluidas ideas como "interrupciones" e incluso "una pila"!

[H74] "Monitores: un concepto de estructuración del sistema operativo" por CAR Hoare. Communications of the ACM, 17:10, páginas 549–557, octubre de 1974. Hoare realizó una buena cantidad de trabajo teórico en paralelo. Sin embargo, probablemente todavía sea más conocido por su trabajo en Quicksort, el algoritmo de clasificación más genial del mundo, al menos según estos autores.

[L11] "Página de manual de señal de condición Pthread" del autor misterioso. Marzo de 2011. Disponible en línea: [http://linux.die.net/man/3/pthread\\_cond\\_signal](http://linux.die.net/man/3/pthread_cond_signal). La página de manual de Linux muestra un ejemplo sencillo y agradable de por qué un hilo puede obtener una reactivación espuria, debido a condiciones de carrera dentro del código de señal/activación.

[LR80] "Experiencia con Procesos y Monitores en Mesa" por BW Lampson, DR Redell. Comunicaciones de la ACM. 23:2, páginas 105-117, febrero de 1980. Un artículo clásico sobre cómo implementar realmente señales y variables de condición en un sistema real, lo que llevó al término semántica de "Mesa" para lo que significa estar despierto; la semántica más antigua, desarrollada por Tony Hoare [H74], pasó a ser conocida como semántica "Hoare", que es un nombre un poco desafortunado.

[O49] "1984" de George Orwell. Secker y Warburg, 1949. Un poco torpe, pero por supuesto una lectura obligada. Dicho esto, de alguna manera revelamos el final al citar la última oración. ¡Lo siento! Y si el gobierno está leyendo esto, digamos simplemente que pensamos que el gobierno es "doble más bueno".

¿Oiste eso, nuestros amigos de la NSA?

## Tarea (Código)

Esta tarea le permite explorar código real que utiliza bloqueos y variables de condición para implementar varias formas de la cola de productor/consumidor que se analiza en el capítulo. Verá el código real, lo ejecutará en varias configuraciones y lo utilizará para aprender qué funciona y qué no, así como otras complejidades. Lea el archivo LÉAME para obtener más detalles.

## Preguntas

1. Nuestra primera pregunta se centra en main-two-cvs- while.c (la solución funcional). Primero, estudia el código. ¿Cree que comprende lo que debería suceder cuando ejecuta el programa?
2. Ejecute con un productor y un consumidor, y haga que el productor produzca algunos valores. Comience con un búfer (tamaño 1) y luego increméntelo. ¿Cómo cambia el comportamiento del código con buffers más grandes? (¿O no?) ¿Qué predeciría que sería num full con diferentes tamaños de búfer (p. ej., -m 10) y diferentes números de artículos producidos (p. ej., -l 100), cuando cambia la cadena de suspensión del consumidor predeterminada (no dormir) a -C 0,0,0,0,0,0,1?
3. Si es posible, ejecute el código en sistemas diferentes (por ejemplo, Mac y Linux). ¿Ves un comportamiento diferente en estos sistemas?
4. Veamos algunos tiempos. ¿Cuánto tiempo cree que tomará la siguiente ejecución, con un productor, tres consumidores, un búfer compartido de entrada única y cada consumidor haciendo una pausa en el punto c3 durante un segundo? ./main-two-cvs-mientras -p 1 -c 3 -m 1 -C 0,0,0,1,0,0,0,0,0,1,0,0,0:0, 0,0,1,0,0,0 -l 10 -v -t
5. Ahora cambie el tamaño del búfer compartido a 3 (-m 3). ¿Esto hará alguna diferencia en el tiempo total?
6. Ahora cambie la ubicación del sueño a c6 (esto modela a un consumidor que toma algo de la cola y luego hace algo con él), nuevamente usando un búfer de entrada única. ¿Qué hora predices en este caso? ./main-two-cvs-mientras -p 1 -c 3 -m 1 -C 0,0,0,0,0,1:0,0,0,0,0,0,1:0, 0,0,0,0,0,1 -l 10 -v -t
7. Finalmente, cambie nuevamente el tamaño del búfer a 3 (-m 3). ¿A qué hora predices ahora?
8. Ahora veamos main-one-cv- while.c. ¿Puedes configurar una cadena de suspensión, asumiendo un único productor, un consumidor y un búfer de tamaño 1, para causar un problema con este código?

9. Ahora cambie el número de consumidores a dos. ¿Puedes construir cadenas de suspensión para el productor y los consumidores de manera que causen un problema en el código?
10. Ahora examine main-two-cvs-if.c. ¿Puedes causar que ocurra un problema en este código? Consideraremos nuevamente el caso en el que hay un solo consumidor y luego el caso en el que hay más de uno.
11. Finalmente, examine main-two-cvs- while-extra-unlock.c. ¿Qué problema surge cuando liberas el bloqueo antes de realizar una venta o una compra? ¿Puedes causar de manera confiable que ocurra tal problema, dadas las cadenas de sueño? ¿Qué cosa mala puede pasar?