2

Introducción a Sistemas Operativos

Si estás cursando Sistemas Operativos en la facultad, ya deberías tener una idea de qué hace un programa de computadora cuando corre. Si no, este libro (y la materia correspondiente) va a ser difícil para vos — así que probablemente deberías dejar de leer este libro, o correr a la librería más cercana y rápidamente consumir el material base antes de continuar (tanto Patt & Patel [PP03], como Bryant & O'Hallaron [BOH10] son muy buenos libros).

Entonces, ¿qué sucede cuando se ejecuta un programa?

Bueno, un programa en ejecución hace una única cosa muy simple: ejecutar instrucciones. Muchas millones (y en la actualidad, incluso miles de millones) de veces por segundo, el procesador **busca** una instrucción en la memoria, la **decodifica** (es decir, descifra cual es la instrucción), y la **ejecuta** (es decir, hace lo que se supone que debería hacer, como sumar dos números entre sí, acceder a la memoria, verificar una condición, saltar a una función, y así). Después de finalizar esta instrucción, el procesador avanza a la siguiente instrucción, y así continúa este procedimiento, hasta que el programa finalmente se completa[1].

Así, hemos descrito lo básico del modelo **Von Newmann**[2]. Suena simple, ¿no? Pero en esta clase, aprenderemos que mientras un programa corre, muchas otras cosas extrañas están ocurriendo las cuales tienen como objetivo primario hacer que el sistema sea **fácil de usar**.

Hay un campo de software que, de hecho, es responsable de hacer que los programas sean fáciles de correr (incluso permitiendo correr

[1] Por supuesto, los procesadores modernos hacen muchas cosas extrañas y espeluznantes detrás de las bambalinas para hacer que los programas corran más rápido: por ejemplo, ejecutar múltiples instrucciones al mismo tiempo, ¡o incluso comenzarlas y completarlas en desorden! Pero no nos preocuparemos de eso aquí; sólo nos interesa el modelo simple que la mayoría de los programas suponen: que las instrucciones parecen ejecutarse una por una, en el orden dado y de manera secuencial.

[2] Von Neumann fue un pionero en sistemas de computación. También realizó un trabajo precursor en la teoría de juegos y bombas atómicas, y jugó en la NBA durante seis años. Bueno, una de esas cosas no es verdad.

EL PROBLEMA EN CUESTIÓN:
CÓMO VIRTUALIZAR RECURSOS

Una pregunta central que vamos a responder en este libro es simple: ¿Cómo hace el sistema operativo para virtualizar recursos? Este es el problema en cuestión. *Por qué* hace esto el SO no es lo principal, ya que la respuesta es obvia: Hace que el sistema sea más fácil de usar. Por lo tanto, nos concentramos en el *cómo*: ¿Qué mecanismos y políticas son implementadas por el SO para alcanzar la virtualización? ¿Cómo lo hace tan eficientemente? ¿Que soporte de hardware es necesario?

Usaremos estas secciones de "El problema en Cuestión" (o simplemente "La Cuestión") como una forma de señalar problemas específicos que estamos intentando solucionar al construir un sistema operativo. Entonces, dentro de la explicación de un tema particular, podés encontrar una o más *cuestiones* que enmarcan el problema. Los detalles del capítulo, por supuesto, presentan la solución, o al menos la idea básica de una solución.

aparentemente muchos programas al mismo tiempo), posibilitando que programas compartan memoria, interactúen con dispositivos, y otras cosas como esa. Este campo de software se llama el **sistema operativo** (**SO**) [3], ya que tiene como objetivo asegurarse que el sistema opere correcta y eficientemente en una forma fácil de usar.

El SO hace esto principalmente mediante una técnica general que llamamos **virtualización**. Esto es, el SO toma un recurso **físico** (como el procesador, la memoria, o un disco) y lo transforma en una versión **virtual** de sí mismo más general, poderosa, y fácil de usar. Por eso, a veces nos referimos al Sistema Operativo como una **máquina virtual**.

Por supuesto, para permitir que usuarios le digan al SO qué hacer y así aprovechar las características de la máquina virtual (como correr un programa, asignar memoria, o acceder a un archivo), el SO provee ciertas interfaces (APIs) que podés llamar. Un SO típico, de hecho, brinda cientos de **llamadas al sistema** disponibles para las aplicaciones. Como el SO provee estas llamadas para correr programas, acceder a memoria y dispositivos, y otras acciones relacionadas, a veces también decimos que el SO provee una **biblioteca estándar** a las aplicaciones.

Finalmente, como la virtualización permite que muchos programas corran (y por lo tanto, compartan la CPU), y que accedan concurrentemente a sus propias instrucciones y datos (por lo tanto, compartiendo memoria), y que accedan a dispositivos (por lo tanto, que

[3] Otro nombre que se le daba al SO era el **Supervisor** o incluso **Programa de Control Maestro**. Aparentemente, el segundo sonaba un poco fanático (para más detalles ver la película Tron) afortunadamente, predominó el uso del término "Sistema Operativo".

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/time.h>
4  #include <assert.h>
5  #include "common.h"
6
7  int
8  main(int argc, char *argv[])
9  {
10     if (argc != 2) {
11         fprintf(stderr, "usage: cpu <string>\n");
12         exit(1);
13     }
14     char *str = argv[1];
15     while (1) {
16         Spin(1);
17         printf("%s\n", str);
18     }
19     return 0;
20 }
```

Figure 2.1: **Simple Ejemplo: Código que itera e imprime (`cpu.c`)**

compartiendo discos y demás), el SO es a veces conocido como un **administrador de recursos**. Cada CPU, memoria, y disco, es un **recurso** del sistema; es entonces el rol del sistema operativo **manejar** esos recursos, haciéndolo eficiente o justo o quizás con muchos otros posibles objetivos en mente. Para entender el rol del SO un poco mejor, veamos algunos ejemplos.

2.1 Virtualizando la CPU

La Figura 2.1 muestra nuestro primer programa. No hace mucho. De hecho, todo lo que hace es llamar `Spin()`, una función que repetidamente verifica la hora y retorna una vez que haya corrido durante un segundo. Luego, imprime la cadena de caracteres que 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 vamos a ver:

```
prompt> gcc -o cpu cpu.c -Wall
prompt> ./cpu "A"
A
A
A
A
```

```
^C
prompt>
```

No es una ejecución demasiado interesante: el sistema comienza a ejecutar el programa, que verifica repetidamente el tiempo hasta que haya transcurrido 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. Tené en cuenta que el programa se ejecutará para siempre; al presionar "Control-c" (que en sistemas basados en UNIX terminará el programa que se ejecuta en primer plano) podemos detener el programa.

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

Figure 2.2: **Corriendo 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 apenas un poco más complicado.

Bueno, ahora las cosas se están poniendo un poco más interesantes. Aunque solo tenemos un procesador, ¡de alguna manera estos cuatro programas parecen estar corriendo al mismo tiempo! ¿Cómo ocurre esta magia? [4]

Resulta que el sistema operativo, con algo de ayuda del hardware, se encarga de esta **ilusión**, es decir, la ilusión de que el sistema tiene una gran cantidad de CPUs virtuales. Convertir una sola CPU (o un pequeño conjunto de ellas) en un número aparentemente infinito de CPU y, por lo tanto, permitir que muchos programas se ejecuten aparentemente a la vez es lo que llamamos **virtualización la CPU**, el foco de la primera parte más importante de este libro. Por supuesto,

[4] Notá cómo ejecutamos cuatro procesos al mismo tiempo, usando el símbolo `&`. Así, se ejecuta un trabajo en segundo plano en la 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 usando una shell diferente (por ejemplo,`tcsh`), funciona de forma ligeramente diferente; leé la documentación online para obtener más detalles.

para ejecutar programas, y detenerlos, y decirle al SO qué programas ejecutar, es necesario que haya algunas interfaces (APIs) que puedas utilizar para comunicarle sus deseos al SO. 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.

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include "common.h"
5
6  int
7  main(int argc, char *argv[])
8  {
9      int *p = malloc(sizeof(int));                    // a1
10     assert(p != NULL);
11     printf("(%d) address pointed to by p: %p\n",
12            getpid(), p);                             // a2
13     *p = 0;                                          // a3
14     while (1) {
15         Spin(1);
16         *p = *p + 1;
17         printf("(%d) p: %d\n", getpid(), *p);        // a4
18     }
19     return 0;
20 }
```

Figure 2.3: **Un Programa que Accede a la Memoria (`mem.c`)**

También puede que hayas notado que la capacidad de ejecutar varios programas a la vez, plantea todo tipo de nuevas preguntas. Por ejemplo, si dos programas quieren ejecutarse en un momento particular, cuál *debería* correr? Esta pregunta es respondida por una **política** del SO; las políticas se utilizan en muchos lugares diferentes dentro de un SO para responder a 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 varios programas a la vez). De ahí el papel del SO como **administrador de recursos**.

2.2 Virtualizando Memoria

Ahora consideremos la memoria. El modelo de **memoria física** presentado por las máquinas modernas es muy simple. La memoria es solo un arreglo de bytes; para **leer** memoria, se debe especificar una **dirección** para poder acceder a los datos almacenados allí; para **escribir** (o **actualizar**) memoria, también se deben especificar los datos que se escribirán en la dirección dada.

Se accede a la memoria todo el tiempo cuando se ejecuta un pro-

grama. Un programa guarda todas sus estructuras de datos en la memoria y accede a ellas a través de varias instrucciones, como ”loads” y ”stores” u otras instrucciones explícitas que acceden a la memoria al realizar su trabajo. No te olvides que cada instrucción del programa también está en la memoria; por lo tanto, se accede a la memoria en cada búsqueda de instrucción.

Miremos un programa (en la Figura 2.3) que asigna algo de memoria al llamar a `malloc()`. La salida de este programa la podés encontrar acá:

```
prompt> ./mem
(2134) address pointed to by p: 0x200000
(2134) p: 1
(2134) p: 2
(2134) p: 3
(2134) p: 4
(2134) p: 5
^C

prompt> ./mem &; ./mem &
[1] 24113
[2] 24114
(24113) address pointed to by p: 0x200000
(24114) address pointed to by 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
...
```

Figure 2.4: **Corriendo Varias Veces el Programa de Memoria**

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 el primer espacio de memoria recién asignada (a3). Finalmente, itera, demorando un segundo e incrementando el valor almacenado en la dirección contenida en `p`. Con cada llamada a print, también imprime lo que se llama el identificador de proceso (el PID) del programa en ejecución. Este PID es único por proceso en ejecución.

Nuevamente, 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` independientemente! 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 [5].

De hecho, eso es exactamente lo que está sucediendo aquí, ya que el SO está **virtualizando la memoria**. Cada proceso accede a su propio espacio de direcciones virtual (a veces simplemente llamado su **espacio de direcciones**), que el SO mapea de alguna manera a la memoria física de la máquina. Una referencia de memoria dentro de un programa en ejecución no afecta el espacio de direcciones de otros procesos (o al SO en sí); 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, administrado por el sistema operativo. Exactamente cómo se logra todo esto es también el tema de la primera parte de este libro, sobre el tema de **virtualización**.

2.3 Concurrencia

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, concurrentemente) en el mismo programa. Los problemas de concurrencia surgieron primero dentro del propio sistema operativo; como podés ver en los ejemplos anteriores sobre virtualización, el SO hace malabares con muchas cosas a la vez, primero ejecuta un proceso, luego otro, y así sucesivamente. Resulta que hacerlo conduce a algunos problemas profundos e interesantes.

Desafortunadamente, los problemas de concurrencia ya no se limitan solo al SO. De hecho, los programas **multi-hilos** modernos presentan los mismos problemas. Dejanos demostrarlo con un ejemplo de un programa **multi-hilo** (Figura 2.5).

Aunque es posible que en este momento no comprendas el ejemplo completamente (y aprenderemos mucho más sobre él en capítulos siguientes, en la sección del libro sobre concurrencia), la idea básica es simple. El programa principal (`main()`) crea dos **hilos** utilizando

[5] Para que este ejemplo funcione, deberías asegurarte de que la aleatorización del espacio de direcciones esté desactivada; la aleatorización, puede ser una buena defensa contra ciertos tipos de fallas de seguridad. Leé más sobre esto por tu cuenta, especialmente si querés aprender cómo ingresar a los sistemas informáticos a través de ataques de stack-smashing. No es que estemos recomendando tal cosa...

```
#include <stdio.h>
#include <stdlib.h>
#include "common.h"
#include "common_threads.h"

volatile int counter = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        counter++;
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: threads <value>\n");
        exit(1);
    }
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", counter);

    Pthread_create(&p1, NULL, worker, NULL);
    Pthread_create(&p2, NULL, worker, NULL);
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("Final value   : %d\n", counter);
    return 0;
}
```

Figure 2.5: **Un Programa Multi-Hilo (`threads.c`)**

`Pthread_create()` [6]. Podés pensar 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 `worker()`, en la que simplemente incrementa un contador en un ciclo durante `loops` veces.

A continuación se muestra una transcripción de lo que sucede cuando ejecutamos este programa con el valor de entrada para la variable `loops` establecida en 1000. El valor de `loops` determina

[6] La llamada real debería estar en minúsculas `pthread_create()`; la versión en mayúsculas es nuestro propio envoltorio que llama a `pthread_create()` y se asegura de que el código de retorno indica que la llamada fue exitosa. Mirar el código para más detalles.

cuántas veces cada uno de los dos *workers* incrementará el contador compartido en un ciclo. Cuando el programa se ejecuta con el valor de `loops` establecido en 1000, ¿Qué valor final de `counter` esperarías?

```
prompt> gcc -o thread thread.c -Wall -pthread
prompt> ./thread 1000
Initial value : 0
Final value   : 2000
```

Como probablemente hayas adivinado, cuando los dos hilos terminaron, el valor final del contador es 2000, ya que cada hilo incrementó el contador 1000 veces. De hecho, cuando el valor de entrada de `loops` se establece en N, esperaríamos que el resultado final del programa fuera $2N$. Pero resulta que la vida no es tan simple. Ejecutemos el mismo programa, pero con valores más altos para `loops` y miremos lo que pasa:

```
prompt> ./thread 100000
Initial value : 0
Final value   : 143012    // huh??
prompt> ./thread 100000
Initial value : 0
Final value   : 137298    // que??
```

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 *distinto* al de la última vez. De hecho, si ejecutaras el programa una y otra vez con valores altos de `loops` ¡Es posible que a veces incluso obtengas la respuesta correcta! Entonces, ¿por qué pasa esto?

EL PROBLEMA EN CUESTIÓN:
CÓMO CONSTRUIR PROGRAMAS CONCURRENTES CORRECTOS

Cuando hay muchos hilos que se ejecutan simultáneamente dentro del mismo espacio de memoria, ¿cómo podemos construir un programa que funcione correctamente? ¿Qué primitivas se necesitan del SO? ¿Qué mecanismos debe proporcionar el hardware? ¿Cómo podemos utilizarlos para resolver los problemas de concurrencia?

Resulta que la razón de estos resultados extraños e inusuales se relaciona con la forma en la que se ejecutan las instrucciones, que es una a la vez. Desafortunadamente, una parte clave del programa anterior, donde se incrementa el contador compartido, toma tres instrucciones: una para cargar el valor del contador de la memoria

en un registro, otra para incrementarlo y otra para almacenarlo nuevamente en la memoria. Como estas tres instrucciones no se ejecutan **atómicamente** (todas a la vez), pueden suceder cosas extrañas. Este es el problema de **concurrencia** que abordaremos muy detalladamente en la segunda parte de este libro.

2.4 Persistencia

El tercer tema del curso es la **persistencia**. En la memoria del sistema, los datos se pueden perder fácilmente, ya que dispositivos como DRAM almacenan valores de manera **volátil**; cuando se produce un corte de energía o el sistema falla, cualquier dato en la memoria se pierde. Por lo tanto, necesitamos hardware y software para poder almacenar datos **persistentemente**; por eso, 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 dispositivo de **entrada/salida** o de **I/O**; en los sistemas modernos, un **disco duro** es un depósito común de información de larga duración, aunque los **discos de estado sólido** (SSD) también están avanzando en este campo.

El software del sistema operativo que generalmente administra el disco se llama **sistema de archivos**; y es responsable de almacenar cualquier **archivo** que el usuario crea, de manera confiable y eficiente en los discos del sistema.

A diferencia de las abstracciones provistas por el SO para la CPU y la memoria, el SO no crea un disco virtual privado para cada aplicación. Más bien, se asume que muchas veces, los usuarios querrán **compartir** información que está en archivos. Por ejemplo, al escribir un programa en C, puede ser que primero uses un editor (por ejemplo, Emacs[7]) para crear y editar el archivo de C (`emacs -nw main.c`). Una vez hecho esto, podés usar el compilador para convertir el código fuente en un ejecutable (por ejemplo, `gcc -o main main.c`). Cuando hayas terminado, podés correr el nuevo ejecutable (por ejemplo, `./main`). Así, podés ver cómo se comparten los archivos entre diferentes procesos. Primero, Emacs crea un archivo que sirve como entrada al compilador; el compilador usa ese archivo de entrada para crear un nuevo archivo ejecutable (en muchos pasos — tomá un curso de compiladores para más detalles); finalmente, se corre el nuevo ejecutable. ¡Y así nace un nuevo programa!

Para entender esto mejor, veamos un código. La Figura 2.6 muestra el código para crear un archivo (`/tmp/file`) que contiene la cadena "hello world".

[7] Deberías estar usando Emacs. Si está utilizando vi, probablemente tengas algún problema. Si estás utilizando algo que no es un editor de código real, es incluso peor.

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <assert.h>
4  #include <fcntl.h>
5  #include <sys/types.h>
6
7  int main(int argc, char *argv[]) {
8      int fd = open("/tmp/file", O_WRONLY|O_CREAT|O_TRUNC,
9                    S_IRWXU);
10     assert(fd > -1);
11     int rc = write(fd, "hello world\n", 13);
12     assert(rc == 13);
13     close(fd);
14     return 0;
15 }
```

Figure 2.6: **Un Programa que Hace I/O (`io.c`)**

Para completar esta tarea, el programa realiza tres llamadas al sistema operativo. El primero, una llamada a `open()`, abre el archivo y lo crea; la segunda, `write()`, escribe algunos datos en el archivo; la tercera, `close()` simplemente cierra el archivo, lo que indica que el programa no escribirá más datos en él. Estás **llamadas al sistema** se enrutan a la parte del sistema operativo denominada **sistema de archivos**, que luego maneja las solicitudes y devuelve algún tipo de código de error al usuario.

Quizás te estás preguntando qué hace el SO para escribir en el disco. Te lo mostraríamos, pero primero tenés que prometer que vas a cerrar los ojos; es así de desagradable. El sistema de archivos tiene que hacer bastante trabajo: primero averiguar en qué parte del disco residirán estos nuevos datos y luego tener un seguimiento de ellos en varias estructuras que tiene el sistema de archivos. Esto requiere emitir solicitudes de I/O al dispositivo de almacenamiento subyacente, para leer las estructuras existentes o actualizarlas (escribirlas). Cualquiera que haya escrito un **controlador de dispositivo** [8] sabe que conseguir que un dispositivo haga algo por vos es un proceso intrincado y detallado. Requiere un conocimiento profundo de la interfaz del dispositivo de bajo nivel y su semántica exacta. Afortunadamente, el SO proporciona una forma estándar y sencilla de acceder a los dispositivos a través de sus llamadas al sistema. Por lo tanto, el SO a veces se ve como una **biblioteca estándar**.

Por supuesto, hay muchos más detalles sobre cómo se accede a los dispositivos y cómo los sistemas de archivos administran los datos de manera persistente sobre dichos dispositivos. Por razones

[8] Un controlador de dispositivo (o en inglés, un device driver) 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 PROBLEMA EN CUESTIÓN:
CÓMO ALMACENAR DATOS PERSISTENTEMENTE
El sistema de archivos es la parte del SO que se encarga de administrar los datos persistentes. ¿Qué técnicas se necesitan para hacerlo correctamente? ¿Qué mecanismos y políticas se requieren para hacerlo con alto desempeño? ¿Cómo se logra la confiabilidad ante fallas en hardware y software?

de rendimiento, la mayoría de los sistemas de archivos primero retrasan dichas escrituras por un tiempo, con la esperanza de juntarlas 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 intrincado, como **journaling** o **copy-on-write**, cuidadosamente ordenar escrituras en el disco para asegurarse de que si ocurre una falla durante la secuencia de escritura, el sistema puede recuperarse a un estado razonable después. Para hacer que las diferentes operaciones comunes sean eficientes, los sistemas de archivos emplean muchas estructuras de datos y métodos de acceso diferentes, desde listas simples hasta complejos árboles binarios. Si todo esto todavía no tiene sentido, ¡bien! Hablaremos de todo esto un poco más en la tercera parte de este libro sobre **persistencia**, donde analizaremos los dispositivos y los I/O en general, y luego los discos, los RAID y los sistemas de archivos muy detalladamente.

2.5 Objetivos de Diseño

Entonces, ahora tenés una idea de lo que realmente hace un SO: toma **recursos** físicos, como una CPU, memoria o disco, y los **virtualiza**. Maneja problemas difíciles y complicados relacionados **concon-currencia**. Y almacena archivos **persistentemente**, lo que los hace seguros a largo plazo. Dado que queremos construir un sistema de este tipo, queremos tener algunos objetivos en mente para ayudar a enfocar nuestro diseño e implementación y hacer negociaciones según sea necesario; encontrar el conjunto correcto de negociaciones es clave para la construcción de sistemas.

Uno de los objetivos más básicos es desarrollar algunas **abstracciones** para que el sistema sea cómodo y fácil de usar. Las abstracciones son fundamentales para todo lo que hacemos en ciencias de la computación. La abstracción hace posible escribir un programa grande dividiéndolo en partes pequeñas y comprensibles, para escribir dicho programa en un lenguaje de alto nivel como C[9] sin pen-

[9]Quizás alguien podría oponerse a llamar a C un lenguaje de alto nivel. Sin em-

sar en assembly, escribir código en assembly sin pensar en compuertas lógicas y construir un procesador a partir de compuertas sin pensar demasiado en transistores. La abstracción es tan fundamental que a veces olvidamos su importancia, pero acá no vamos a descuidarlo; por eso, en cada sección, discutiremos algunas de las principales abstracciones que se han desarrollado a lo largo del tiempo, lo que te brindará una forma de pensar las partes del SO.

Uno de los objetivos del diseño e implementación de un sistema operativo es proporcionar alto rendimiento; otra forma de decir esto es que nuestro objetivo es **minimizar los gastos generales** del SO. La virtualización y hacer que el sistema sea fácil de usar valen la pena, pero no a cualquier costo; por lo tanto, debemos esforzarnos por proporcionar virtualización y otras características del SO sin gastos excesivos. Estos gastos surgen de varias formas: tiempo adicional (más instrucciones) y espacio adicional (en la memoria o en el disco). Buscaremos soluciones que minimicen uno u otro, o ambos, si es posible. La perfección, sin embargo, no siempre es alcanzable, algo que aprenderemos a distinguir y (cuando sea apropiado) a tolerar.

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

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 el campo (incluido parte de nuestro propio trabajo [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 ecológico; la **seguridad** (una extensión de la protección, en realidad) contra aplicaciones maliciosas es fundamental, especialmente en estos tiempos con tanta conectividad a la red; la **movilidad** es cada vez más importante a medida que los sistemas operativos se ejecutan en dispositivos cada vez más

bargo, no te olvides que este es un curso de SO, en el que simplemente estamos felices de no tener que programar en assembly todo el tiempo.

pequeños. Dependiendo de cómo se use el sistema, el SO tendrá diferentes objetivos y, por lo tanto, es probable que se implemente al menos de formas ligeramente diferentes. Sin embargo, como veremos, muchos de los principios que presentaremos sobre cómo construir un SO son útiles en un muchos 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 seres humanos, las buenas ideas se acumularon en los sistemas operativos a lo largo del tiempo, a medida que los ingenieros e ingenieras aprendieron lo que era importante en su diseño. Aquí, discutimos algunos desarrollos importantes. Para un enfoque más rico, consultá la excelente historia de sistemas operativos de Brinch Hansen [BH00].

Los Primeros Sistemas Operativos: Solo Bibliotecas

Al principio, el sistema operativo no hacía demasiado. Básicamente, era solo un conjunto de bibliotecas de funciones de uso común; por ejemplo, en lugar de tener que escribir el código de manejo de I/O de bajo nivel, el "SO" proporcionaba dichas APIs y, por lo tanto, facilitaba la vida del desarrollador o desarrolladora.

Por lo general, en estos viejos sistemas de mainframe, se ejecutaba un programa a la vez, controlado por una persona que hacía de operador. Gran parte de lo que pensás que haría un sistema operativo moderno (por ejemplo, decidir en qué orden ejecutar los trabajos) lo realizaba esta persona operadora. Si fueras una o un programador inteligente, serías amable con ella, para que pudieran mover tu trabajo al principio de la cola.

Este modo de computación se conocía como procesamiento por **lote**, ya que quien operaba 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 dejar que usuarios se sentaran frente a la computadora y la usaran, ya que la mayoría de las veces simplemente se quedaba inactiva en ese momento, lo que costaba 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 administración de máquinas. Un aspecto importante fue la comprensión

de que el código que se ejecuta para del sistema operativo era especial; tenía el control de los dispositivos y, por tanto, debería tratarse de forma diferente al código de una aplicación normal. ¿Por qué es esto? Bueno, imaginá si se permitiera que cualquier aplicación lea desde cualquier lugar del disco; la noción de privacidad se va por la ventana, 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.

Por tanto, se inventó la idea de una **llamada al sistema**, siendo pionero el sistema informático Atlas [K+61,L78]. En lugar de proporcionar rutinas del SO como una biblioteca (donde simplemente crea una l**lamada de procedimiento** para acceder a ellos), la idea aquí era agregar un par especial de instrucciones de hardware y un estado de hardware para hacer de la transición al SO un proceso más formal y controlado.

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 SO mientras que simultáneamente aumenta el **nivel de privilegio de hardware**. Las aplicaciones de usuario se ejecutan en lo que se denomina **modo usuario** lo que significa que el hardware restringe lo que pueden hacer las aplicaciones; por ejemplo, una aplicación que se ejecuta en modo usuario normalmente no puede iniciar una solicitud de I/O al disco, acceder a cualquier página de memoria física o enviar un paquete a la red. Cuando se inicia una llamada al sistema (generalmente a través de una instrucción de hardware especial llamada **trap**), el hardware transfiere el control a un **trap-handler** (que se encarga de las traps, configurado previamente por el sistema operativo) y simultáneamente eleva el nivel de privilegio a **modo kernel**. En el modo kernel, el SO tiene acceso a todo el hardware del sistema y, por lo tanto, puede hacer cosas como iniciar una solicitud de I/O o hacer que haya más memoria disponible para un programa. Cuando el SO termina de atender la solicitud, devuelve el control al usuario a través de una instrucción de **regreso del trap**, que vuelve al modo usuario y, al mismo tiempo, pasa el control en donde la aplicación había dejado.

La Era de la Multiprogramación

Cuando los sistemas operativos realmente despegaron fue en la era de la computación más allá del mainframe, la era de la **minicomputadora**. Las máquinas clásicas como la familia PDP de Digital Equipment hicieron que las computadoras fueran mucho más accesibles; por lo tanto, en lugar de tener un mainframe por organización grande, ahora un grupo más pequeño de personas dentro de una organización probablemente podría tener su propia computadora. Como era de esperar, uno de los principales impactos de esta caída

en el costo fue un aumento en la actividad en el campo de la programación; más gente inteligente puso sus manos en las computadoras y así hizo que los sistemas informáticos hicieran cosas más interesantes y hermosas.

En particular, la **multiprogramación** se convirtió en algo común debido al deseo de hacer un mejor uso de los recursos de la máquina. En lugar de ejecutar un trabajo a la vez, el SO carga varios trabajos en la memoria y cambia rápidamente entre ellos, mejorando así la utilización de la CPU. Estos cambios entre procesos fueron particularmente importantes porque los dispositivos de I/O eran lentos; tener un programa esperando en la CPU mientras se atendía un I/O era una pérdida de tiempo de la CPU. En su lugar, ¿por qué no cambiar a otro trabajo y ejecutarlo durante un tiempo?

El deseo de soportar la multiprogramación y la superposición en presencia de I/O y las interrupciónes forzaron la innovación en el desarrollo conceptual de los sistemas operativos en de varias direcciones. Cuestiones como la **protección de la memoria** se volvieron importantes; no querríamos que un programa pudiera acceder a la memoria de otro programa. Entender cómo lidiar con los problemas de **concurrencia** introducidos por la multiprogramación también fue crítico; asegurarse de que el SO se comporte correctamente a pesar de la presencia de interrupciones es un gran desafío. Estudiaremos estos problemas y temas relacionados más adelante en el libro.

Uno de los principales avances prácticos de la época fue la introducción de la sistema operativo UNIX, principalmente gracias a Ken Thompson (y Dennis Ritchie) en 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 los hizo más simples y fáciles de usar. Pronto este equipo estaba enviando cintas que contenían código fuente de UNIX a personas de todo el mundo, muchas de las cuales luego se involucraron y contribuyeron al sistema; ver **Aparte** (siguiente página) para más detalles[10].

La Era Moderna

Más allá de la minicomputadora llegó un nuevo tipo de máquina, más barata, más rápida y para las masas: la **computadora personal**, o **PC** como lo llamamos hoy. Liderada por las primeras máquinas de Apple (por ejemplo, la Apple II) y la PC de IBM, esta nueva generación de máquinas pronto se convertiría en la fuerza dominante en la informática, ya que su bajo costo posibilitaba una máquina por es-

[10] Usaremos apartes y otros cuadros de texto relacionados para llamar la atención sobre varios temas que no encajan en el flujo principal del texto. A veces, incluso los usamos solo para hacer una broma, porque ¿por qué no divertirnos un poco en el camino? Sí, 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 original "Bell Labs" UNIX era el principio unificador de la creación de pequeños programas potentes que se podían conectar entre sí para formar flujos de trabajo más grandes. La **shell**, donde escribís comandos, proporcionó primitivas como **pipes** (tuberías) para habilitar programación de meta-nivel, y así se hizo fácil encadenar programas para lograr una tarea más grande. Por ejemplo, para buscar líneas de un archivo de texto que contengan la palabra "foo" y luego contar cuántas de esas líneas existen, deberías escribir: `grep foo file.txt | wc -l`, así usando los programas `grep` y `wc` (word count, recuento de palabras) relizás tu tarea.

El entorno UNIX era amigable tanto para las personas programadoras como para las desarrolladoras, y también proporcionaba un compilador para el nuevo **Lenguaje de programación C**. Facilitar a quienes programaban escribir sus propios programas, así como compartirlos, hizo que UNIX fuera enormemente popular. Y probablemente ayudó mucho que los y las autoras entregaran copias gratis a cualquiera que las solicitara, una forma temprana de **software de código abierto**.

También fue de vital importancia la accesibilidad y legibilidad del código. Tener un kernel pequeño y hermoso escrito en C invitaba a otros a jugar con el kernel, agregando características nuevas y geniales. Por ejemplo, un grupo emprendedor en Berkeley, dirigido por **Bill Joy**, hizo una distribución maravillosa (**Berkeley Systems Distribution**, o **BSD**) que tenía memoria virtual avanzada, sistema de archivos y subsistemas de redes. Joy luego co-fundó **Sun Microsystems**.

Desafortunadamente, la propagación de UNIX se ralentizó un poco a medida que las empresas intentaban afirmar la propiedad y beneficiarse de ella, un resultado desafortunado (pero común) de la participación de abogados/as. Muchas empresas tenían sus propias variantes: **SunOS** variante de Sun Microsystems, **AIX** de IBM, **HPUX** (también conocido como "H-Pucks") de HP, y **IRIX** de SGI. Las disputas legales entre AT&T/Bell Labs y estos otros jugadores crearon 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 PCs...

critorio en lugar de una minicomputadora compartida por grupo de trabajo.

APARTE: Y LUEGO VINO LINUX

Afortunadamente para UNIX, un joven hacker finlandés llamado **Linus Torvalds** decidió escribir su propia versión de UNIX que tomó prestados, en gran medida, los principios e ideas detrás del sistema original, pero no de la base del código, evitando así problemas legales. Consiguió la ayuda de muchos otros en todo el mundo, aprovechó las sofisticadas herramientas GNU que ya existían [G85], y pronto **Linux** nació (así como el movimiento moderno del software de código abierto).

Cuando llegó la era de Internet, la mayoría de las empresas (como Google, Amazon, Facebook y otros) eligieron ejecutar Linux, ya que era gratuito y podía modificarse fácilmente para adaptarse a sus necesidades; de hecho, es difícil imaginar el éxito que estas nuevas empresas habrían tenido si no existiera tal sistema. A medida que los teléfonos inteligentes se convirtieron en una plataforma dominante oriantada al usuario, Linux también encontró un punto fuerte allí (a través de Android), por muchas de las mismas razones. Y Steve Jobs se llevó su entorno operativo **NeXTStep** basado en UNIX a Apple, lo que hace que UNIX sea popular en las computadoras de escritorio (aunque muchos usuarios de Apple probablemente ni siquiera sean conscientes de este hecho). Por lo tanto, UNIX vive, más importante hoy que nunca. Las deidades de la computación, si crees en ellas, deben ser agradecidas por este maravilloso resultado.

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** (**Disk Operating Sistem**, de Microsoft) no creían que la protección de la memoria fuera importante; por lo tanto, una aplicación maliciosa (o tal vez simplemente mal programada) podría hacer garabatos en toda la memoria. Las primeras generaciones de **Mac OS** (v9 y anteriores) adoptaron un enfoque cooperativo para la planificación de trabajos; por lo tanto, un hilo que se trabara accidentalmente en un ciclo infinito podría tomar el control de todo el sistema, forzando un reinicio. La dolorosa lista de características del sistema operativo que faltan en esta generación de sistemas es larga, demasiado larga para una discusión completa aquí.

Afortunadamente, después de algunos años de sufrimiento, las viejas características de los sistemas operativos de minicomputadoras comenzaron a abrirse camino en el escritorio. Por ejemplo, Mac OS X/macOS tiene UNIX en su núcleo, incluidas todas las características que cabría esperar de un sistema tan maduro. Windows ha adoptado de manera similar muchas de las grandes ideas de la histo-

ria de la informática, comenzando en particular con Windows NT, un gran avance en la tecnología del SO Microsoft. Incluso los teléfonos móviles actuales ejecutan sistemas operativos (como Linux) que se parecen más a lo que funcionaba una minicomputadora en la década de 1970 que a lo que corría una PC en la década de 1980 (gracias a Dios); es bueno ver que las buenas ideas desarrolladas en el apogeo del desarrollo de SO se han abierto camino en el 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

Entonces, tenemos una introducción al SO. Los sistemas operativos actuales hacen que los sistemas sean relativamente fáciles de usar, y prácticamente todos los sistemas operativos que usamos hoy se han visto influenciados por los desarrollos que discutiremos a lo largo del libro.

Desafortunadamente, debido a limitaciones de tiempo, hay una serie de partes del SO que no cubriremos en el libro. Por ejemplo, hay mucho código de **redes** en el sistema operativo; te dejamos tomar la clase de redes para aprender más sobre eso. De manera similar, los dispositivos de **gráficos** son particularmente importantes; tomá el curso de gráficos para ampliar sus conocimientos en esa dirección. Por último, algunos libros de sistemas operativos hablan mucho sobre **seguridad**; lo haremos en el sentido de que el SO debe brindar protección entre los programas en ejecución y brindarle a usuarios la capacidad de proteger sus archivos, pero no ahondaremos en los 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, concurrencia y persistencia a través de dispositivos y sistemas de archivos. ¡No te preocupes! Si bien hay mucho terreno por recorrer, la mayor parte es bastante interesante y, al final del camino, tendrás una nueva apreciación de cómo funcionan realmente los sistemas informáticos. ¡Ahora, manos a la obra!

Referencias

[BS+09] "Tolerating File-System Mistakes with EnvyFS" by L. Bairavasundaram, S. Sundararaman, A. Arpaci-Dusseau, R. Arpaci-Dusseau. USENIX '09, San Diego, CA, June 2009. *Un artículo divertido sobre el uso de varios sistemas de archivos a la vez para tolerar un error en cualquiera de ellos.*

[BH00] "The Evolution of Operating Systems" by P. Brinch Hansen. In 'Classic Operating Systems: From Batch Processing to Distributed Systems.' Springer-Verlag, New York, 2000. *Este ensayo proporciona una introducción a una maravillosa colección de artículos sobre sistemas históricamente significativos.*

[B+72] "TENEX, A Paged Time Sharing System for the PDP-10" by D. Bobrow, J. Burchfiel, D. Murphy, R. Tomlinson. CACM, Volume 15, Number 3, March 1972. *TENEX tiene gran parte de la maquinaria que se encuentra en los sistemas operativos modernos; leé más sobre él para ver cuánta innovación ya existía a principios de la década de 1970.*

[B75] "The Mythical Man-Month" by F. Brooks. Addison-Wesley, 1975. *Un texto clásico sobre ingeniería de software; vale la pena leerlo.*

[BOH10] "Computer Systems: A Programmer's Perspective" by R. Bryant and D. O'Hallaron. Addison-Wesley, 2010. *Otra gran introducción a cómo funcionan los sistemas informáticos. Tiene un poco de superposición con este libro, por lo que, si querés, podés 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 tu propio conocimiento es escuchar tantas otras perspectivas como sea posible y luego desarrollar tu propia opinión sobre el tema. ¡Ya sabes, pensando!*

[G85] "The GNU Manifesto" by R. Stallman. 1985. `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 cuando se trata de código abierto, y este manifiesto expone sus pensamientos sobre el por qué.*

[K+61] "One-Level Storage System" by T. Kilburn, D.B.G. Edwards, M.J. Lanigan, F.H. Sumner. IRE Transactions on Electronic Computers, April 1962. *Atlas fue pionero en gran parte de lo que se ve en los sistemas modernos. Sin embargo, este documento no es la mejor lectura. Si solo tuvieras que leer uno, podrías probar "A Historical Perspective" [L78].*

[L78] "The Manchester Mark I and Atlas: A Historical Perspective" by S. H. Lavington. Communications of the ACM, Volume 21:1, January 1978. *Una buena parte de la historia sobre el desarrollo de los sistemas informáticos y los esfuerzos pioneros del Atlas. Por supuesto, uno podría volver atrás y leer los propios artículos del Atlas, pero este artículo proporciona una gran descripción general y agrega cierta perspectiva histórica.*

[O72] "The Multics System: An Examination of its Structure" by Elliott Organick. MIT Press, 1972. *Una gran descripción de Multics. Tantas buenas ideas y, sin embargo, era un sistema sobre-diseñado, ambicionando demasiado y, por lo tanto, nunca funcionó realmente. Un ejemplo clásico de lo que Fred Brooks llamaría el "efecto de segundo sistema" [B75].*

[PP03] "Introduction to Computing Systems: From Bits and Gates to C and Beyond" by Yale N. Patt, Sanjay J. Patel. McGraw-Hill, 2003. *Uno de nuestros libros favoritos de introducción a los sistemas informáticos. Comienza en transistores y llega hasta C; el material inicial es particularmente bueno.*

[RT74] "The UNIX Time-Sharing System" by Dennis M. Ritchie, Ken Thompson. CACM, Volume 17: 7, July 1974. *Un gran resumen de UNIX escrito mientras se estaba apoderando del mundo de la informática, por las personas que lo escribieron*

[S68] "SDS 940 Time-Sharing System" by Scientific Data Systems. TECHNICAL MANUAL, SDS 90 11168, August 1968. *.Sí, lo mejor que pudimos encontrar fue un manual técnico. Pero es fascinante leer estos documentos del antiguo sistema y ver cuánto ya existía a fines de la década de 1960. Una de las mentes detrás del Berkeley Time-Sharing System (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: Operating System Support for Restartable File Systems" by S. Sundararaman, S. Subramanian, A. Rajimwale, A. Arpaci-Dusseau, R. Arpaci-Dusseau, M. Swift. FAST '10, San Jose, CA, February 2010. *Lo mejor de escribir tus propias notas de clase: puedes publicitar tu propia investigación. Pero este documento es bastante bueno — cuando un sistema de archivos detecta un error y crashea, Membrane lo reinicia automágicamente, todo sin que las aplicaciones o 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 te permite a vos, el lector o lectora, adquirir más experiencia con los conceptos presentados en el capítulo.

Hay dos tipos de tareas. El primero se basa en **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 reporta en la salida algunas métricas para mostrar cómo se comporta el sistema. Por ejemplo, un simulador de disco duro podría tomar una serie de solicitudes, simular cuánto 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 correr un sistema real. De hecho, incluso te 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 tecnologías futuras.

Obviamente, las simulaciones no están exentas de desventajas. Por su propia naturaleza, las simulaciones son solo aproximaciones de cómo se comporta un sistema real. Si se omite un aspecto importante del comportamiento del mundo real, la simulación reportará malos resultados. Por lo 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 solo requieren un desarrollo y experimentación a pequeña escala. Ambos son solo pequeñas incursiones en el mundo más grande en el que deberías estar ingresando, que es cómo escribir código de sistemas en C en sistemas basados en UNIX. De hecho, se necesitan proyectos a mayor escala, que van más allá de hacer estas tareas, para mejorar en esta dirección; por lo tanto, más allá de hacer las tareas, te recomendamos encarecidamente que realices proyectos para solidificar sus habilidades en sistemas. Ver esta pagina para encontrar algunos proyectos

`https://github.com/remziarpacidusseau/ostep-projects`

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