Interfaz de usuario gráfica, Sitio web

Descripción generada automáticamente**INSTITUTO POLITÉCNICO NACIONAL**

**ESCUELA SUPERIOR DE CÓMPUTO**

INGENIERÍA EN SISTEMAS COMPUTACIONALES

**INVESTIGACIÓN DOCUMENTAL**

**UNIDAD DE APRENDIZAJE:**

SISTEMAS OPERATIVOS

**PROFESOR:**

SALAS RAMÍREZ ISRAEL

**EQUIPOS:**

EQUIPO 404 Y ERROR DE CAPA 8

**ALUMNOS:**

CASIMIRO HERNÁNDEZ GAEL NOEL

CONDE GALVÁN LESLIE MONSERRAT

MÁRQUEZ HERNÁNDEZ SARAH MICHELLE

MARTÍNEZ LEÓN JOSUÉ DAVID

MEJÍA ALMAZÁN GERARDO NICOLAS

**GRUPO:**

4CV1

**FECHA DE ENTREGA:**

11 / 06 / 2025

**ÍNDICE**

[**UNIDAD TEMÁTICA 6 SEGURIDAD Y VIRTUALIZACIÓN** 1](#_Toc200811999)

[**6.1) EL AMBIENTE DE SEGURIDAD** 1](#_Toc200812000)

[**6.1.1) SEGURIDAD EN LOS SISTEMAS OPERATIVOS** 2](#_Toc200812001)

[**6.1.2) CONTROL DE ACCESO A LOS RECURSOS** 5](#_Toc200812002)

[**6.1.3) IMPLEMENTACIÓN DE MATRICES DE ACCESO** 6](#_Toc200812003)

[**6.1.4) MODELOS FORMALES DE SEGURIDAD** 11](#_Toc200812004)

[**ACTIVIDAD DE APRENDIZAJE 1: FUNDAMENTOS DE SEGURIDAD** 14](#_Toc200812005)

[**6.2) VIRTUALIZACIÓN** 17](#_Toc200812006)

[**6.2.1) EMULACIÓN** 20](#_Toc200812007)

[**6.2.2) VIRTUALIZACIÓN ASISTIDA POR HARDWARE** 22](#_Toc200812008)

[**6.2.3) PARAVIRTUALIZACIÓN** 24](#_Toc200812009)

[**6.2.4) CONTENEDORES** 34](#_Toc200812010)

[**ACTIVIDAD DE APRENDIZAJE 2:** **VIRTUALIZACIÓN** 46](#_Toc200812011)

[**REFERENCIAS** 48](#_Toc200812012)

**TABLA DE TABLAS**

[Tabla 1. Tipos de amenazas de seguridad en sistemas computacionales (adaptado de Silberschatz et al., 2018). 1](#_Toc200812014)

[Tabla 2. Tipos de matrices y vectores para la gestión de recursos (basado en Carretero et al., 2001). 6](#_Toc200812015)

[Tabla 3. Matriz C de asignación (ejemplo del libro Sistemas Operativos: Una visión aplicada de Jesús Carretero). 7](#_Toc200812016)

[Tabla 4. Matriz D de solicitudes (ejemplo del libro Sistemas Operativos: Una visión aplicada de Jesús Carretero). 7](#_Toc200812017)

[Tabla 5. Componentes principales del modelo Clark-Wilson (basado en Stallings, 2018). 12](#_Toc200812018)

[Tabla 6. Integración de modelos de seguridad con hardware moderno (adaptado de Silberschatz et al., 2018). 13](#_Toc200812019)

[Tabla 7. Características fundamentales de la emulación (basado en Tanenbaum & Bos, 2015). 20](#_Toc200812020)

[Tabla 8. Tipos de emulación y sus aplicaciones (adaptado de Stallings, 2018; Wolf et al., 2015). 21](#_Toc200812021)

[Tabla 9. Componentes de virtualización asistida por hardware (basado en Tanenbaum & Bos, 2015; Stallings, 2018). 22](#_Toc200812022)

[Tabla 10. Comparativa de rendimiento: software vs. hardware-assisted virtualization (adaptado de Silberschatz et al., 2018). 23](#_Toc200812023)

[Tabla 11. Comparación entre Virtualización Tradicional y Paravirtualización (elaboración propia basada en Wolf et al., 2015; Tanenbaum & Bos, 2015). 25](#_Toc200812024)

[Tabla 12. Comparativa de Rendimiento: Paravirtualización vs. Virtualización Tradicional (adaptado de Kivity et al., 2007; Russell, 2008; Rizzo et al., 2012). 27](#_Toc200812025)

[Tabla 13. Viabilidad de Paravirtualización por Tipo de Sistema Operativo (elaboración propia basada en Wolf et al., 2015). 28](#_Toc200812026)

[Tabla 14. Evolución Histórica de las Tecnologías de Contenedores. 35](#_Toc200812027)

[Tabla 15. Comparación Arquitectónica: Contenedores vs. Otros Métodos de Virtualización (elaboración propia basada en Wolf et al., 2015; Merkel, 2014). 37](#_Toc200812028)

[Tabla 16. Tipos de Recursos Gestionados en Contenedores (elaboración propia basada en Wolf et al., 2015; Merkel, 2014). 40](#_Toc200812029)

[Tabla 17. Comparativa de Eficiencia: Contenedores vs. Virtualización Tradicional (adaptado de Merkel, 2014; Felter et al., 2015; Sharma et al., 2016). 41](#_Toc200812030)

[Tabla 18. Variaciones en Implementaciones de Contenedores (elaboración propia basada en Wolf et al., 2015; FreeBSD Wiki, 2025). 42](#_Toc200812031)

**TABLA DE FIGURAS**

[Figura 1. Arquitectura de Comunicación en Paravirtualización. 26](#_Toc200812032)

[Figura 2. Arquitectura de Paravirtualización Híbrida. 29](#_Toc200812033)

[Figura 3. Arquitectura VMI para Múltiples Hipervisores. 31](#_Toc200812034)

[Figura 4. Arquitectura de Aislamiento en Contenedores. 38](#_Toc200812035)

**TABLA DE CÓDIGOS**

[Código 1. Simulación de matrices de acceso para prevención de deadlocks. 8](#_Toc200812037)

[Código 2. Estructura simplificada de una entrada de tabla de páginas en sistemas de 32 bits. 18](#_Toc200812038)

[Código 3. Ejemplo de implementación de operaciones paravirtualizadas de E/S. 33](#_Toc200812039)

[Código 4. Ejemplo simplificado de implementación de contenedores usando namespaces de Linux. 45](#_Toc200812040)

# **UNIDAD TEMÁTICA 6 SEGURIDAD Y VIRTUALIZACIÓN**

## **6.1) EL AMBIENTE DE SEGURIDAD**

#### **INTRODUCCIÓN**

En los últimos años, el acceso a la red y diferentes tipos de computadoras para acceder a internet ha permitido que los ataques del exterior, comúnmente conocidos como virus, gusanos, troyanos entre otros, pudieran entrar y causar estragos en los dispositivos comunes [Silberschatz et al., 2018]. Esto aunado con el crecimiento del bugware inflado (partes de código que son difíciles de reemplazar y su funcionalidad no está especificada), había sustituido el software eficiente de años anteriores [Tanenbaum & Bos, 2015].

Con el crecimiento en funciones del sistema operativo también lo hace el código en el kernel, con más de 5 millones de líneas de código actualmente se vuelve difícil dar mantenimiento y revisar errores más a detalle [Stallings, 2018]. Para empezar este capítulo daremos una definición sobre los términos de "seguridad" y "protección" que, aunque parezcan sinónimos, no lo son, ambos tienen diferencias en cómo actúa cada uno.

Un ejemplo claro es distinguir sobre los problemas generales involucrados en el proceso que personas no autorizadas lean o modifiquen archivos, lo que nos lleva a cuestiones técnicas administrativas, legales, y de políticas que se manejen (seguridad), por otro lado, están los mecanismos específicos del sistema operativo que llevará a cabo la tarea de brindar tal seguridad (protección) [Carretero et al., 2001].

Desde la perspectiva de la seguridad en sistemas computacionales se tienen cuatro objetivos generales con las correspondientes amenazas, como se presenta en la Tabla 1.

|  |  |
| --- | --- |
| **OBJETIVO** | **AMENAZA** |
| Confidencialidad de datos. | Exposición de datos. |
| Integridad de los datos. | Alteración de los datos. |
| Disponibilidad del sistema. | Negación del servicio. |
| Exclusión de los usuarios externos. | Los virus se apropian del sistema. |

Tabla . Tipos de amenazas de seguridad en sistemas computacionales (adaptado de Silberschatz et al., 2018).

### **6.1.1) SEGURIDAD EN LOS SISTEMAS OPERATIVOS**

#### **INTRODUCCIÓN**

La seguridad en un sistema dentro de una de sus facetas incluye la protección de los datos, desde los posibles daños físicos (fuegos, terremotos, o cualquier situación externa) hasta el acceso indebido a los mismos (corrupción de los datos, intrusos, fallos en la confidencialidad, etc.) [Stallings, 2018]. Frente a un ataque contra la integridad o la disponibilidad de los datos en un sistema operativo debe haber un plan de contingencia para la prevención o la evasión del ataque [Tanenbaum & Bos, 2015].

Para el caso de un sistema informático, hay una diversidad de elementos susceptibles de sufrir dichos ataques, siendo suficiente con proteger sólo algunos de ellos o proteger en su totalidad a todos los elementos parcialmente [Silberschatz et al., 2018].

La protección por su parte consiste en evitar que se haga uso indebido de los recursos que se encuentran dentro del ámbito de un sistema informático [Carretero et al., 2001]. Por esto mismo deben existir mecanismos y políticas que aseguren que cada usuario va a acceder únicamente a sus propios recursos (siendo éstos archivos, zonas de memoria, etc.). Además, se debe asegurar que los recursos sólo se usan por usuarios que tienen derechos de acceso a los mismos [Stallings, 2018]. Las políticas de protección y seguridad de hardware, software y datos deben incluirse dentro del sistema operativo [Carretero et al., 2001].

#### **PROBLEMAS DE SEGURIDAD**

Hasta la fecha la mayor parte de los sistemas operativos desarrollados para su uso comercial han tenido algún problema de seguridad [Tanenbaum & Bos, 2015]. Muchos de los fallos no se deben al sistema de protección, sino a la simpleza del usuario al poner como palabra de seguridad al sistema el nombre de algún conocido cercano o algún número como fechas o matrículas [Silberschatz et al., 2018]. A continuación, se describen los problemas más frecuentes en un sistema informático:

**USO INDEBIDO O MALICIOSO DE PROGRAMAS**

Algunos problemas de seguridad están relacionados con el uso indebido, o malicioso de programas de usuario [Stallings, 2018]. Usualmente, dos de las formas más frecuentes de generar fallos de seguridad en un sistema informático son usando las técnicas conocidas como el *caballo de Troya* y la *puerta de atrás* [Tanenbaum & Bos, 2015].

La técnica de El caballo de Troya se basa en la idea de crear un programa para que haga cosas no autorizadas en el sistema cuando actúa, es decir, se ejecuta en el entorno adecuado [Silberschatz et al., 2018]. Mientras que la técnica de La puerta de atrás consiste en crear un agujero de seguridad al sistema a través de un programa al que se le dé un permiso de ejecución con permisos de un administrador [Carretero et al., 2001]. Básicamente se basan en la creación de programas que faciliten el acceso de algún intruso mediante la persuasión al usuario de filtrar credenciales o que den permisos de acceso a programas "no dañinos" [Stallings, 2018].

**USUARIOS NO AUTORIZADOS**

En algunos sistemas operativos, como lo son UNIX o Windows, se mantiene el uso de cuentas para los usuarios no autorizados o que cuentan con menos permisos de seguridad [Tanenbaum & Bos, 2015]. El acceso a estas cuentas se protege mediante contraseñas o palabras de seguridad, las cuales usualmente sólo conoce el dueño de dichas cuentas [Silberschatz et al., 2018]. Este dueño o responsable es comúnmente denominado como administrador o súper-usuario, ya que, puede acceder a todos los recursos del sistema saltándose las protecciones del resto de los usuarios [Carretero et al., 2001].

Para que se pueda llevar a cabo la modificación o la ejecución de un programa en la computadora es necesario acceder a una cuenta de usuario y en caso de ser necesario, aprobar la prueba de autenticación de un administrador [Stallings, 2018].

**VIRUS**

Comúnmente se denomina con este nombre a todos los programas que se autorreplican con fines destructivos o de violación de seguridad [Tanenbaum & Bos, 2015]. Su nombre se deriva por la analogía con los virus que actúan sobre los seres vivos, ya que, de igual forma necesitan un medio que transporte el virus y de un agente transmisor, en este caso los programas infectan a otros programas [Silberschatz et al., 2018].

En todos los casos se trata de un programa o trozo de un código que se añade a otro y que se activa cuando se ejecuta el programa portador del virus [Stallings, 2018].

**GUSANOS**

Se trata de un programa que se encarga de explotar los fallos de seguridad para acceder a sistemas remotos y, además, que se replique a sí mismo para acceder a más sistemas [Tanenbaum & Bos, 2015]. A comparación de los virus, los gusanos no necesitan un programa anfitrión para propagarse, sino que pueden autorreplicarse a través de la red [Silberschatz et al., 2018].

**BOMBARDEO**

Se entiende como bombardeo a un ataque de seguridad en el que consiste en llevar a cabo bombardeos masivos con peticiones de servicio o de establecimiento de conexión a un servidor determinado, en este caso el servidor al que se va a atacar [Tanenbaum & Bos, 2015]. Estos ataques masivos hacen que el servidor deniegue sus servicios a los clientes legales, además, los bombardeos pueden llegar a bloquear un servidor que esté siendo víctima del mismo [Silberschatz et al., 2018].

### **6.1.2) CONTROL DE ACCESO A LOS RECURSOS**

#### **INTRODUCCIÓN**

En el ámbito interno de un sistema operativo hay operaciones que pueden violar la confidencialidad de los datos [Carretero et al., 2001]. Para evitar que algún usuario tenga acceso a datos que no son suyos, el sistema operativo debe encargarse de limpiar los recursos de los datos anteriormente existentes [Stallings, 2018].

El control del acceso a datos y recursos suele ser una competencia directa en el sistema operativo [Silberschatz et al., 2018]. Debido a que es necesario asegurar que los usuarios no acceden a archivos para los que no tienen permisos de acceso, a cuentas de otros usuarios o a páginas de memoria o bloques de disco que contienen información de otros usuarios [Tanenbaum & Bos, 2015].

Para lograr esto hay que aplicar criterios de diseño rigurosos y ejecutar pruebas de seguridad exhaustivas para todos los elementos del sistema, aunque se consideren seguros [Carretero et al., 2001]. El control de acceso incluye dos problemas principales: la autenticación de usuarios, y la protección frente accesos indebidos [Stallings, 2018].

### **6.1.3) IMPLEMENTACIÓN DE MATRICES DE ACCESO**

#### **INTRODUCCIÓN**

Las matrices de acceso representan una herramienta indispensable en la administración de recursos dentro de los sistemas operativos [Carretero et al., 2001]. Permiten modelar de manera clara la relación entre procesos y recursos, facilitando la detección y prevención de interbloqueos (deadlocks), así como la asignación segura y eficiente de los recursos del sistema [Silberschatz et al., 2018].

Estas estructuras son fundamentales para garantizar el correcto funcionamiento del sistema, especialmente en entornos multitarea o de tiempo compartido [Tanenbaum & Bos, 2015]. En el libro *Sistemas Operativos: Una visión aplicada* de Jesús Carretero Pérez y colaboradores, se detalla su utilidad para evitar estados inseguros en la asignación de recursos [Carretero et al., 2001].

**CONCEPTO Y ESTRUCTURA DE LA MATRIZ DE ACCESO**

Una matriz de acceso es una estructura de datos bidimensional que representa las asignaciones y solicitudes de recursos por parte de los procesos [Stallings, 2018]. Se consideran matrices auxiliares para completar la gestión de recursos, como se describe en la Tabla 2:

|  |  |  |
| --- | --- | --- |
| **TIPO DE MATRIZ** | **DESCRIPCIÓN** | **PROPÓSITO** |
| Matriz de asignación (C) | Contiene cuántas unidades de cada tipo de recurso están asignadas a cada proceso | Control de recursos actualmente en uso |
| Matriz de solicitudes pendientes (D) | Lleva el control de cuántos recursos ha pedido cada proceso y que aún no se le han asignado | Gestión de peticiones en espera |
| Vector de recursos disponibles (E) | Indica cuántas unidades libres hay de cada recurso en ese momento | Monitoreo de recursos libres |
| Vector de recursos totales (A) | Cantidad total que el sistema tiene disponible de cada tipo de recurso | Capacidad máxima del sistema |

Tabla . Tipos de matrices y vectores para la gestión de recursos (basado en Carretero et al., 2001).

Un ejemplo práctico que se encuentra en el libro Sistemas Operativos: Una visión aplicada, de Jesús Carretero Pérez y colaboradores, considera el siguiente escenario:

**Procesos:** P1, P2, P3

**Recursos:** R1 (3 unidades), R2 (5 unidades), R3 (2 unidades)

**MATRIZ C (ASIGNACIÓN)**

|  |  |  |  |
| --- | --- | --- | --- |
| **PROCESO** | **R1** | **R2** | **R3** |
| P1 | 1 | 2 | 0 |
| P2 | 0 | 1 | 1 |
| P3 | 2 | 0 | 1 |

Tabla . Matriz C de asignación (ejemplo del libro Sistemas Operativos: Una visión aplicada de Jesús Carretero).

**MATRIZ D (SOLICITUDES)**

|  |  |  |  |
| --- | --- | --- | --- |
| **PROCESO** | **R1** | **R2** | **R3** |
| P1 | 1 | 0 | 1 |
| P2 | 2 | 1 | 0 |
| P3 | 0 | 1 | 0 |

Tabla . Matriz D de solicitudes (ejemplo del libro Sistemas Operativos: Una visión aplicada de Jesús Carretero).

**Vector E (Disponibles):** [0, 1, 0]

**Vector A (Totales):** [3, 5, 2]

En este caso, podemos ver que no hay unidades disponibles de R1 ni de R3, lo que puede llegar a complicar el proceso si intenta pedir de más [Carretero et al., 2001].

Es posible programar una simulación sencilla de cómo funcionan esas matrices. Si lo implementamos a código obtenemos el siguiente Código 1 en C:

#include <stdio.h>

#define PROCESOS 3

#define RECURSOS 3

int main() {

int asignacion[PROCESOS][RECURSOS] = {

{1, 2, 0},

{0, 1, 1},

{2, 0, 1}

};

int solicitudes[PROCESOS][RECURSOS] = {

{1, 0, 1},

{2, 1, 0},

{0, 1, 0}

};

int disponibles[RECURSOS] = {0, 1, 0};

int terminados[PROCESOS] = {0};

int trabajo[RECURSOS];

for (int i = 0; i < RECURSOS; i++) {

trabajo[i] = disponibles[i];

}

int progreso = 1;

while (progreso) {

progreso = 0;

for (int i = 0; i < PROCESOS; i++) {

if (!terminados[i]) {

int puede\_terminar = 1;

for (int j = 0; j < RECURSOS; j++) {

if (solicitudes[i][j] > trabajo[j]) {

puede\_terminar = 0;

break;

}

}

if (puede\_terminar) {

printf("Proceso P%d puede terminar.\n", i+1);

for (int j = 0; j < RECURSOS; j++) {

trabajo[j] += asignacion[i][j];

}

terminados[i] = 1;

progreso = 1;

}

}

}

}

return 0;

}

Código . Simulación de matrices de acceso para prevención de deadlocks.

Este código analiza si los procesos pueden completarse con los recursos disponibles, liberando lo que usan una vez que terminan [Silberschatz et al., 2018]. Así se evita que el sistema entre en un estado "inseguro" [Tanenbaum & Bos, 2015].

**APLICACIÓN EN LA PREVENCIÓN DE LOS DEADLOCKS**

Según Carretero et al. (2001), el sistema operativo puede usar estas matrices para hacer simulaciones antes de asignar un recurso. Por lo tanto, si al hacer la simulación se detecta que el estado resultante sería inseguro (es decir, que podría haber procesos que nunca terminen), entonces este rechaza la solicitud [Stallings, 2018]. De esta manera se previenen posibles interbloqueos antes de que ocurran.

Por ejemplo, si el proceso P2 pide más unidades y el sistema se da cuenta de que eso dejaría sin posibilidad de ejecución a los demás procesos, lo mejor es no darle los recursos por ese momento [Silberschatz et al., 2018].

Por lo tanto, concluimos que las matrices de acceso son una forma práctica de gestionar recursos en los sistemas operativos [Carretero et al., 2001]. Ayudan a prevenir errores críticos como los interbloqueos [Tanenbaum & Bos, 2015].

### **6.1.4) MODELOS FORMALES DE SEGURIDAD**

#### **INTRODUCCIÓN**

Los modelos formales de seguridad son como el marco teórico que necesitamos para asegurar que nuestros datos e información estén protegidos dentro de los sistemas operativos [Stallings, 2018]. Estos modelos definen reglas matemáticas estrictas sobre quién puede acceder a qué, asegurando la confidencialidad y la integridad de la información [Silberschatz et al., 2018].

Gracias a ellos, se pueden implementar controles efectivos en sistemas con múltiples usuarios [Tanenbaum & Bos, 2015]. A continuación, vamos a explorar los modelos más relevantes que se describen en el libro *Sistemas Operativos: Una Visión Aplicada* [Carretero et al., 2001], entendiendo sus principios, estructuras y cómo se aplican en situaciones prácticas.

#### **MODELO DE SEGURIDAD BELL-LAPADULA PROTEGIENDO LA CONFIDENCIALIDAD**

El modelo Bell-LaPadula fue pensado inicialmente para entornos militares y se enfoca en prevenir la fuga de información [Stallings, 2018]. Tiene dos reglas clave:

* **Propiedad de seguridad simple:** Un proceso no puede leer datos de un nivel de seguridad más alto que el suyo (esto se llama no read up) [Silberschatz et al., 2018].
* **Propiedad -estrella:** Un proceso no puede escribir datos en un nivel de seguridad más bajo que el suyo (esto se conoce como no write down) [Tanenbaum & Bos, 2015].

Este modelo utiliza etiquetas de seguridad como "Confidencial", "Secreto" o "Ultra Secreto" para clasificar tanto a los usuarios como a los archivos [Carretero et al., 2001]. Por ejemplo, un agente con acceso "Secreto" no puede ver documentos "Ultra Secreto" ni modificar archivos "Confidenciales" desde su nivel [Stallings, 2018].

#### **MODELO BIBA PROTEGIENDO LA INTEGRIDAD**

A diferencia de Bell-LaPadula, el modelo Biba se centra en mantener la integridad de los datos [Silberschatz et al., 2018]. Aquí, las reglas son las siguientes:

* **Propiedad de integridad simple:** Un proceso no puede escribir datos en un nivel superior de lo que tiene permitido (no write up) [Tanenbaum & Bos, 2015].
* **Propiedad de invocación:** Un proceso no puede recibir información de un nivel inferior al suyo [Stallings, 2018].

Este modelo es clave en sistemas donde la exactitud de los datos es crucial, como en sistemas bancarios [Carretero et al., 2001]. Por ejemplo, un cajero automático no puede modificar registros financieros de nivel superior, como los de auditoría [Silberschatz et al., 2018].

#### **MODELO DE CLARK-WILSON MANTENIENDO LA INTEGRIDAD EN EL COMERCIO**

Clark-Wilson introduce conceptos nuevos que se presentan en la Tabla 5:

|  |  |  |
| --- | --- | --- |
| **CONCEPTO** | **DESCRIPCIÓN** | **EJEMPLO DE APLICACIÓN** |
| Procedimientos certificados (TPs) | Operaciones validadas que garantizan la integridad | Transferencias bancarias |
| Entidades restringidas (CDIs) | Datos que solo se pueden modificar mediante TPs | Registros de ensayos clínicos |

Tabla . Componentes principales del modelo Clark-Wilson (basado en Stallings, 2018).

Este modelo es común en sistemas comerciales donde los procesos deben seguir reglas predefinidas [Tanenbaum & Bos, 2015]. Por ejemplo, en una farmacéutica, solo se permiten ciertos procedimientos validados para actualizar registros de ensayos clínicos [Silberschatz et al., 2018].

#### **MODELO DAC CONTROL DISCRECIONAL DE ACCESO**

El modelo de Control de Acceso Discrecional (DAC) permite que el propietario de un recurso decida quién puede acceder a él [Carretero et al., 2001]. Se utiliza a través de:

* **Listas de Control de Acceso (ACLs):** Especifican permisos como lectura, escritura o ejecución [Stallings, 2018].
* **Capacidades:** Tokens que otorgan derechos específicos [Silberschatz et al., 2018].

Por ejemplo, en sistemas UNIX, el modelo DAC es utilizado con comandos como chmod 755 archivo.txt, donde el dueño tiene control total y otros usuarios solo pueden leer o ejecutar el archivo [Tanenbaum & Bos, 2015].

#### **CONSIDERACIONES EN SISTEMAS OPERATIVOS MODERNOS**

En los sistemas actuales, se combinan estos modelos con características de hardware, como se muestra en la Tabla 6:

|  |  |  |
| --- | --- | --- |
| **TECNOLOGÍA** | **PROPÓSITO** | **EJEMPLO DE IMPLEMENTACIÓN** |
| Anillos de protección | Diferentes niveles de privilegio | Anillo 0 para el núcleo, anillo 3 para aplicaciones |
| Unidades de Gestión de Memoria (MMU) | Aislamiento de procesos | Tablas de páginas separadas |
| Virtualización Basada en Seguridad (VBS) | Entornos aislados | Servicios críticos de Windows |

Tabla . Integración de modelos de seguridad con hardware moderno (adaptado de Silberschatz et al., 2018).

En sistemas como Windows, el modelo Biba se integra con la Virtualización Basada en Seguridad (VBS) para crear entornos aislados para servicios críticos [Stallings, 2018].

### **ACTIVIDAD DE APRENDIZAJE 1: FUNDAMENTOS DE SEGURIDAD**

Para cada uno de los reactivos, selecciona la opción correcta al enunciado.

1. ¿Cuál es la principal diferencia entre seguridad y protección en sistemas operativos?

a) Seguridad se refiere a hardware, protección a software

b) No hay diferencia, son sinónimos

c) Seguridad incluye políticas administrativas y legales, protección son mecanismos específicos del SO

**d) Seguridad es para usuarios, protección para administradores**

2. En el modelo Bell-LaPadula, la propiedad \*-estrella establece que:

a) Se puede leer información de niveles superiores

b) Se puede escribir en cualquier nivel

c) No se puede escribir información en niveles inferiores

**d) No se puede leer información de niveles inferiores**

3. Las matrices de acceso se utilizan principalmente para:

a) Acelerar el procesamiento

b) Gestionar la memoria virtual

c) Prevenir interbloqueos y gestionar recursos

**d) Controlar dispositivos de entrada/salida**

4. El modelo Biba se enfoca principalmente en:

a) Confidencialidad de datos

b) Integridad de datos

c) Disponibilidad del sistema

**d) Autenticación de usuarios**

5. Un caballo de Troya se caracteriza por:

a) Autorreplicarse sin programa anfitrión

b) Atacar servidores masivamente

c) Hacer cosas no autorizadas cuando actúa en el entorno adecuado

**d) Adivinar contraseñas de usuarios**

6. ¿Qué característica distingue principalmente a un gusano de un virus informático?

a) Solo afecta a sistemas UNIX.

b) Siempre requiere la intervención del usuario para activarse.

c) Su único propósito es borrar archivos del disco duro.

**d) No necesita un programa anfitrión para propagarse y puede autorreplicarse a través de la red.**

7. El control de acceso en un sistema operativo aborda principalmente dos problemas. ¿Cuáles son?

a) La velocidad de la red y el espacio en disco.

b) La compatibilidad del software y la gestión de energía.

c) El cifrado de datos y la compresión de archivos.

**d) La autenticación de usuarios y la protección frente a accesos indebidos.**

8. En el modelo de seguridad Biba, ¿qué establece la "propiedad de integridad simple"?

a) Un proceso puede leer datos de cualquier nivel de integridad.

b) Un proceso no puede leer datos de un nivel inferior al suyo.

c) Un proceso puede escribir datos en cualquier nivel de integridad.

**d) Un proceso no puede escribir datos en un nivel de integridad superior al suyo.**

9. ¿Cuál es el objetivo de un ataque de "bombardeo" o denegación de servicio (DoS)?

a) Robar las contraseñas de los usuarios del servidor.

b) Instalar un caballo de Troya en el sistema.

c) Modificar los datos almacenados en el servidor.

**d) Provocar que el servidor deniegue sus servicios a clientes legítimos y eventualmente se bloquee.**

10. ¿Cuál es el principio fundamental del modelo de Control de Acceso Discrecional (DAC)?

a) El acceso es controlado por etiquetas de seguridad definidas por el sistema.

b) Todos los usuarios tienen los mismos permisos por defecto.

c) Solo el administrador del sistema puede asignar permisos.

**d) El propietario de un recurso es quien decide y gestiona los permisos de acceso sobre él.**

## **6.2) VIRTUALIZACIÓN**

#### **INTRODUCCIÓN**

En el estudio de los sistemas operativos, la virtualización es uno de los conceptos clave que nos ayuda a entender cómo un sistema puede ofrecer eficiencia, seguridad y flexibilidad al manejar sus recursos [Tanenbaum & Bos, 2015]. Una de las formas más claras de aplicar la virtualización es en el uso de la memoria virtual, que permite que los programas se ejecuten como si tuvieran toda la memoria para sí mismos, cuando en realidad están compartiendo un recurso limitado con otros procesos [Silberschatz et al., 2018].

#### **¿QUÉ ES LA MEMORIA VIRTUAL?**

La memoria virtual les permite a los programas pensar que tienen acceso a un espacio de memoria exclusivo y continuo, incluso si la memoria física no es suficiente o está siendo compartida [Stallings, 2018]. Sus principales objetivos son:

* Ejecutar programas grandes: Aprovechar mejor la RAM cuando no es suficiente.
* Aislar los procesos: Evitar que los programas interfieran entre sí.
* Mejorar la multiprogramación: Hacer que los recursos se usen al máximo, cargando solo lo que realmente se necesite.

**¿CÓMO FUNCIONA?**

El sistema operativo implementa la virtualización mediante varios componentes [Tanenbaum & Bos, 2015]:

* Traducción de direcciones: La MMU (Unidad de Gestión de Memoria) convierte las direcciones virtuales en físicas utilizando las tablas de páginas.
* Paginación bajo demanda: Solo se cargan en la RAM las páginas que se necesitan y las demás se guardan en el disco.
* Espacio de intercambio (swap): Un área en el disco que guarda las páginas que no están activas, es útil cuando la RAM se llena.

**¿CÓMO FUNCIONA LA MMU Y LAS TABLAS DE PÁGINAS?**

La MMU utiliza las tablas jerárquicas para convertir direcciones [Silberschatz et al., 2018]. Por ejemplo, en un sistema de 32 bits, la estructura de una entrada de la tabla de páginas sería como se muestra en el Código 2:

// Estructura simplificada de una entrada de tabla de páginas

struct page\_table\_entry {

uint32\_t physical\_frame : 20; // Marco físico

uint32\_t present : 1; // Bit de presencia

uint32\_t writable : 1; // Permiso de escritura

// ... otros flags (acceso, modificado)

};

Código . Estructura simplificada de una entrada de tabla de páginas en sistemas de 32 bits.

Cuando un proceso trata de acceder a una dirección no cargada en la RAM (bit de presencia=0), el sistema genera un "fallo de página" y comienza el proceso de carga desde el disco duro [Stallings, 2018].

#### **EJEMPLO PRÁCTICO ¿QUÉ PASA CUANDO EJECUTAMOS VARIOS PROGRAMAS A LA VEZ?**

Imagina que tienes una computadora con 2 GB de memoria RAM. Sin memoria virtual, si intentas ejecutar un programa que necesita 1.5 GB y otro que necesita 1 GB, simplemente no podrías correr ambos al mismo tiempo [Tanenbaum & Bos, 2015]. Pero con memoria virtual, el sistema operativo te permite hacerlo gracias a varios trucos inteligentes:

1. Espacios de memoria virtual por proceso: Cada programa que se ejecuta recibe su propio espacio de direcciones virtuales. Por ejemplo, se les puede asignar 4 GB de espacio virtual, aunque físicamente no estén disponibles [Silberschatz et al., 2018].
2. Traducción de direcciones: Cuando un programa intenta acceder a una dirección de memoria, esa dirección no se refiere directamente a la RAM física. El sistema operativo, con ayuda de la MMU, traduce esa dirección virtual a una dirección física mediante tablas de páginas [Stallings, 2018].
3. El disco duro como memoria de apoyo: Cuando ya no hay suficiente espacio en la RAM para todo lo que se está usando, el sistema operativo puede mover temporalmente partes de los programas al disco duro, en un archivo llamado swap o de intercambio [Tanenbaum & Bos, 2015].

Este ejemplo de memoria virtual muestra perfectamente cómo funciona la virtualización en los sistemas operativos. No solo permite que varios programas se ejecuten al mismo tiempo, sino que también protege los datos de cada uno y hace un uso más inteligente de la memoria [Silberschatz et al., 2018].

### **6.2.1) EMULACIÓN**

#### **INTRODUCCIÓN**

La emulación representa uno de los enfoques fundamentales en la virtualización de sistemas computacionales, permitiendo que un sistema (el anfitrión) ejecute software diseñado para una arquitectura completamente diferente (el huésped) [Tanenbaum & Bos, 2015]. A diferencia de otras técnicas de virtualización que aprovechan similitudes de hardware, la emulación puede crear un entorno completamente artificial que simula el comportamiento de hardware inexistente en el sistema físico [Stallings, 2018].

La emulación funciona mediante la creación de una capa de software que traduce cada instrucción del sistema huésped en una o más instrucciones equivalentes del sistema anfitrión [Silberschatz et al., 2018]. Este proceso, aunque computacionalmente costoso, proporciona una compatibilidad total que permite ejecutar sistemas operativos y aplicaciones sin modificación alguna [Wolf et al., 2015].

#### **CARACTERÍSTICAS FUNDAMENTALES**

Las principales características de la emulación se resumen en la Tabla 7:

|  |  |  |  |
| --- | --- | --- | --- |
| **CARACTERÍSTICA** | **DESCRIPCIÓN** | **VENTAJA** | **DESVENTAJA** |
| Compatibilidad total | No requiere modificaciones del software huésped | Ejecuta cualquier SO sin cambios | Alto overhead computacional |
| Independencia de arquitectura | Puede emular hardware completamente diferente | Máxima flexibilidad | Rendimiento significativamente reducido |
| Traducción de instrucciones | Cada instrucción se traduce individualmente | Precisión en la emulación | Latencia en la ejecución |
| Simulación completa | Emula todos los componentes del sistema | Funcionalidad completa | Uso intensivo de recursos |

Tabla . Características fundamentales de la emulación (basado en Tanenbaum & Bos, 2015).

#### **TIPOS DE EMULACIÓN**

Existen diferentes enfoques para implementar emulación, cada uno con sus propias ventajas y casos de uso específicos, como se presenta en la Tabla 8:

|  |  |  |  |
| --- | --- | --- | --- |
| **TIPO DE EMULACIÓN** | **DESCRIPCIÓN** | **EJEMPLOS** | **CASOS DE USO** |
| Emulación de hardware | Simula componentes físicos específicos | QEMU, VirtualBox | Testing, desarrollo, legacy systems |
| Emulación de consolas | Replica sistemas de videojuegos | MAME, Dolphin | Preservación digital, gaming |
| Emulación de CPU | Traduce instrucciones de procesador | Rosetta (Apple), Dynamic recompilation | Migración entre arquitecturas |
| Emulación de sistema completo | Simula todo el sistema computacional | VMware, Parallels | Múltiples SO en una máquina |

Tabla . Tipos de emulación y sus aplicaciones (adaptado de Stallings, 2018; Wolf et al., 2015).

#### **VENTAJAS Y LIMITACIONES**

La emulación ofrece beneficios únicos, pero también presenta desafíos significativos. Sus principales ventajas incluyen la capacidad de ejecutar software legacy, realizar testing en múltiples plataformas sin hardware físico, y preservar sistemas históricos [Carretero et al., 2001]. Sin embargo, el costo computacional puede ser de 10 a 100 veces mayor que la ejecución nativa, dependiendo de la complejidad de la traducción requerida [Silberschatz et al., 2018].

### **6.2.2) VIRTUALIZACIÓN ASISTIDA POR HARDWARE**

#### **INTRODUCCIÓN**

La virtualización asistida por hardware representa una evolución significativa en las tecnologías de virtualización, introduciendo capacidades específicas en los procesadores para facilitar y acelerar la creación y gestión de máquinas virtuales [Tanenbaum & Bos, 2015]. Esta aproximación busca resolver las limitaciones de rendimiento inherentes a la virtualización por software mediante la implementación de instrucciones especializadas y mecanismos de hardware dedicados [Stallings, 2018].

Las tecnologías modernas como Intel VT-x (Virtualization Technology) y AMD-V (AMD Virtualization) han transformado el panorama de la virtualización al proporcionar soporte nativo en el procesador para la ejecución de múltiples sistemas operativos [Silberschatz et al., 2018]. Estas extensiones permiten que el hypervisor opere con mayor eficiencia, reduciendo el overhead tradicionalmente asociado con la virtualización [Wolf et al., 2015].

#### **COMPONENTES**

Los elementos clave que componen la virtualización asistida por hardware se detallan en la Tabla 9:

|  |  |  |  |
| --- | --- | --- | --- |
| **COMPONENTE** | **FUNCIÓN** | **TECNOLOGÍA INTEL** | **TECNOLOGÍA AMD** |
| Extensiones de CPU | Soporte nativo para hypervisors | VT-x | AMD-V |
| Gestión de memoria | Traducción de direcciones virtualizadas | EPT (Extended Page Tables) | NPT (Nested Page Tables) |
| Manejo de I/O | Virtualización de dispositivos | VT-d | AMD-Vi |
| Interrupciones | Gestión eficiente de interrupciones | VT-x with APIC | AVIC (Advanced Virtual Interrupt Controller) |

Tabla . Componentes de virtualización asistida por hardware (basado en Tanenbaum & Bos, 2015; Stallings, 2018).

#### **BENEFICIOS DE RENDIMIENTO**

La virtualización asistida por hardware ofrece mejoras sustanciales en rendimiento comparada con enfoques puramente software, como se muestra en la Tabla 10:

|  |  |  |  |
| --- | --- | --- | --- |
| **MÉTRICA** | **VIRTUALIZACIÓN SOFTWARE** | **HARDWARE-ASSISTED** | **MEJORA** |
| Overhead de CPU | 15-30% | 2-5% | 75-85% |
| Gestión de memoria | Alto costo de traducción | Hardware nativo | 90-95% |
| Cambios de contexto | Emulación completa | Instrucciones dedicadas | 80-90% |
| Manejo de interrupciones | Software trap and emulate | Hardware virtualization | 70-80% |

Tabla . Comparativa de rendimiento: software vs. hardware-assisted virtualization (adaptado de Silberschatz et al., 2018).

#### **ARQUITECTURA DE FUNCIONAMIENTO**

El funcionamiento de la virtualización asistida por hardware se basa en la creación de dos modos de operación distintos: el modo raíz (root mode) donde opera el hypervisor, y el modo no-raíz (non-root mode) donde ejecutan las máquinas virtuales [Carretero et al., 2001]. Esta separación permite que el hardware gestione automáticamente las transiciones entre el hypervisor y las VMs, reduciendo significativamente el overhead de las operaciones de virtualización [Wolf et al., 2015].

### **6.2.3) PARAVIRTUALIZACIÓN**

#### **INTRODUCCIÓN**

La paravirtualización representa una evolución significativa en las tecnologías de virtualización, emergiendo como una respuesta directa a las limitaciones inherentes de los métodos tradicionales de emulación y virtualización asistida por hardware. A diferencia de estos enfoques, donde el sistema operativo huésped permanece inconsciente de su entorno virtualizado, la paravirtualización adopta un paradigma completamente diferente basado en la cooperación y el conocimiento mutuo entre el hipervisor y los sistemas operativos invitados.

La paravirtualización consiste en alojar sistemas operativos huésped que, a sabiendas de que están ejecutando en hardware virtualizado, no hacen llamadas directas a hardware, sino que las traducen a llamadas al sistema operativo anfitrión [Wolf et al., 2015]. Este enfoque fundamental transforma la naturaleza misma de la virtualización, pasando de un modelo de engaño y emulación a uno de colaboración consciente.

Como establece Tanenbaum y Bos (2015), "un sistema operativo invitado para el que se han eliminado de manera intencional (algunas) instrucciones sensibles está paravirtualizado" (p. 574). Esta definición aparentemente simple encapsula un cambio paradigmático fundamental en cómo conceptualizamos la virtualización.

#### **PRINCIPIOS FUNDAMENTALES DE LA PARAVIRTUALIZACIÓN**

**DEFINICIÓN Y ARQUITECTURA BÁSICA**

La paravirtualización se fundamenta en un principio revolucionario: eliminar la necesidad de emulación completa del hardware mediante la modificación consciente del sistema operativo huésped. El hipervisor debe definir una interfaz que consiste en un conjunto de llamadas a procedimientos que los sistemas operativos invitados puedan utilizar, formando efectivamente una API (Interfaz de Programación de Aplicaciones) específicamente diseñada para sistemas operativos y no para aplicaciones convencionales [Tanenbaum & Bos, 2015].

La Tabla 11 presenta una comparación detallada entre la virtualización tradicional y la paravirtualización, destacando las diferencias arquitectónicas fundamentales.

|  |  |  |
| --- | --- | --- |
| **ASPECTO** | **VIRTUALIZACIÓN TRADICIONAL** | **PARAVIRTUALIZACIÓN** |
| Conocimiento del entorno | SO desconoce la virtualización | SO consciente de la virtualización |
| Modificación del código | No requiere cambios | Requiere modificación del código fuente |
| Tipo de llamadas al sistema | Llamadas directas emuladas | Llamadas específicas al hipervisor |
| Overhead computacional | Alto debido a la emulación | Significativamente reducido |
| Complejidad de implementación | Menor para el SO huésped | Mayor para el SO huésped |
| Estabilidad del sistema | Susceptible a fallos de emulación | Mayor estabilidad por simplicidad |

Tabla . Comparación entre Virtualización Tradicional y Paravirtualización (elaboración propia basada en Wolf et al., 2015; Tanenbaum & Bos, 2015).

El proceso de transformación de un sistema operativo tradicional a uno paravirtualizado puede ser equivalente a adecuar al sistema operativo para que ejecute en una arquitectura nueva [Wolf et al., 2015]. Esta analogía ilustra la magnitud del cambio requerido: no se trata simplemente de ajustes menores, sino de una reingeniería fundamental de cómo el sistema operativo interactúa con el hardware subyacente.

**ARQUITECTURA DE COMUNICACIÓN**

La arquitectura de la paravirtualización se construye sobre una interfaz bien definida entre el hipervisor y los sistemas operativos invitados [Tanenbaum & Bos, 2015]. Los controladores de hardware tradicionales se convierten casi en simples pasarelas de llamadas al sistema, brindando además de una sobrecarga mínima, aún mayor estabilidad por simplicidad del código [Wolf et al., 2015].

La Figura 1 ilustra la arquitectura de comunicación en entornos paravirtualizados, mostrando cómo se establece la comunicación directa entre el sistema operativo huésped y el hipervisor.

Diagrama

El contenido generado por IA puede ser incorrecto.

Figura . Arquitectura de Comunicación en Paravirtualización.

#### **VENTAJAS Y BENEFICIOS OPERACIONALES**

**MEJORAS SIGNIFICATIVAS EN RENDIMIENTO**

La paravirtualización ofrece mejoras de rendimiento documentadas y significativas en comparación con las técnicas de virtualización tradicionales [Wolf et al., 2015]. Según mediciones empíricas realizadas por Qumranet en 2007, las clases de dispositivos virtio y pv resultaron entre 5 y 10 veces más rápidas que la emulación de dispositivos reales [Wolf et al., 2015].

La Tabla 12 presenta una comparativa detallada del rendimiento entre diferentes métodos de virtualización.

|  |  |  |  |
| --- | --- | --- | --- |
| **MÉTRICA** | **VIRTUALIZACIÓN TRADICIONAL** | **PARAVIRTUALIZACIÓN** | **MEJORA** |
| Operaciones de E/S por segundo | Línea base (100%) | 500-1000% | 5-10x |
| Latencia de red | Alta (emulación completa) | Baja (llamadas directas) | 70-80% |
| Overhead de CPU | 15-25% | 3-5% | 75-85% |
| Throughput de disco | Limitado por emulación | Cercano al nativo | 300-400% |
| Uso de memoria | Alto (buffers de emulación) | Optimizado | 40-60% |

Tabla . Comparativa de Rendimiento: Paravirtualización vs. Virtualización Tradicional (adaptado de Kivity et al., 2007; Russell, 2008; Rizzo et al., 2012).

**ESTABILIDAD Y CONFIABILIDAD MEJORADAS**

La eliminación de la capa de emulación no solo mejora el rendimiento, sino que también incrementa significativamente la estabilidad del sistema [Silberschatz et al., 2018]. Al reducir la complejidad del código que maneja las operaciones críticas del sistema, la paravirtualización minimiza los puntos de fallo potenciales [Silberschatz et al., 2018].

#### **PARAVIRTUALIZACIÓN Y SOFTWARE LIBRE**

**REQUISITOS DE ACCESO AL CÓDIGO FUENTE**

Una característica fundamental de la paravirtualización es su dependencia del acceso al código fuente del sistema operativo [Wolf et al., 2015]. Para que los autores de un entorno que implemente paravirtualización logren que un sistema operativo nuevo pueda ser ejecutado en su arquitectura, deben poder manipular y modificar su código fuente [Wolf et al., 2015].

La Tabla 13 muestra la viabilidad de paravirtualización según el tipo de sistema operativo y el acceso al código fuente.

|  |  |  |  |
| --- | --- | --- | --- |
| **TIPO DE SISTEMA** | **ACCESO AL CÓDIGO** | **VIABILIDAD** | **EJEMPLOS** |
| Software Libre | Completo | Alta | Linux, FreeBSD, OpenSolaris |
| Software Propietario | Limitado/Nulo | Baja | Windows, macOS |
| Sistemas Académicos | Bajo licencias especiales | Media | Windows XP (Academic Licensing) |
| Sistemas Embebidos | Variable | Media-Alta | Sistemas IoT, RTOS |

Tabla . Viabilidad de Paravirtualización por Tipo de Sistema Operativo (elaboración propia basada en Wolf et al., 2015).

**EL CASO PARADIGMÁTICO DE XEN**

El proyecto Xen, nacido como una iniciativa académica de la Universidad de Cambridge, ilustra perfectamente tanto las posibilidades como las limitaciones de la paravirtualización. Según Barham et al. (2003), Xen sólo pudo ser empleado por muchos años como plataforma de paravirtualización de Linux, mientras que las versiones paravirtualizadas de sistemas propietarios permanecieron inaccesibles para el público general debido a restricciones de licenciamiento [Wolf et al., 2015].

#### **PARAVIRTUALIZACIÓN DE DISPOSITIVOS**

**IMPLEMENTACIÓN EN ENTORNOS HÍBRIDOS**

Una de las innovaciones más significativas en el campo de la paravirtualización es la posibilidad de implementar paravirtualización parcial de dispositivos dentro de entornos de virtualización completa [Wolf et al., 2015]. Si el sistema operativo está estructurado de una forma modular, no hace falta modificar al sistema operativo completo para gozar de los beneficios de la paravirtualización en algunas áreas [Wolf et al., 2015].

La Figura 2 ilustra la arquitectura de paravirtualización híbrida, donde ciertos componentes utilizan paravirtualización mientras otros mantienen emulación tradicional.

Diagrama

El contenido generado por IA puede ser incorrecto.

Figura . Arquitectura de Paravirtualización Híbrida.

Esta aproximación híbrida permite que varios aspectos que son parte del núcleo duro del sistema, como la administración de memoria o el manejo de interrupciones, tendrán que seguirse manejando mediante una emulación, aunque mucho más delgada, mientras que los dispositivos de alta actividad se benefician de la eficiencia de la paravirtualización [Wolf et al., 2015].

**DISPOSITIVOS OBJETIVO PRIORITARIOS**

Los dispositivos que más se benefician de la paravirtualización son aquellos que típicamente generan más actividad de entrada y salida [Wolf et al., 2015]. Estos incluyen:

* **Medios de almacenamiento:** Discos duros, SSDs, sistemas de archivos distribuidos
* **Interfaces de red:** Tarjetas Ethernet, interfaces inalámbricas, conexiones de alta velocidad
* **Salida de video:** Aceleración gráfica, renderizado 3D, procesamiento de video

#### **INTERFAZ DE MÁQUINA VIRTUAL (VMI)**

**SOLUCIÓN AL PROBLEMA DE PORTABILIDAD**

La proliferación de diferentes hipervisores con APIs incompatibles creó un desafío significativo para la adopción generalizada de la paravirtualización [Tanenbaum & Bos, 2015]. ¿Qué pasa si hay varios hipervisores disponibles en el mercado como VMware, el Xen de código fuente abierto y Microsoft Viridian, todos ellos con APIs de hipervisor algo distintas? [Tanenbaum & Bos, 2015]. La solución propuesta por Amsden y colaboradores introdujo el concepto de VMI (Virtual Machine Interface) [Tanenbaum & Bos, 2015]. El kernel se modifica para llamar a ciertos procedimientos especiales cada vez que necesita hacer algo sensible, creando una capa de abstracción que permite la portabilidad entre diferentes plataformas de virtualización [Tanenbaum & Bos, 2015].

La Figura 3 muestra la arquitectura VMI para múltiples hipervisores, demonstrando cómo un mismo sistema operativo puede ejecutarse en diferentes plataformas.

Diagrama

El contenido generado por IA puede ser incorrecto.

Figura . Arquitectura VMI para Múltiples Hipervisores.

**IMPLEMENTACIONES ALTERNATIVAS**

Además del VMI, han surgido otras técnicas para abordar el problema de la interfaz de máquina virtual [Tanenbaum & Bos, 2015]. Una de las más populares es paravirt ops. Esta idea es similar en concepto a lo que hemos visto antes, pero difiere en algunos detalles [Tanenbaum & Bos, 2015].

#### **EJEMPLO PRÁCTICO: IMPLEMENTACIÓN DE PARAVIRTUALIZACIÓN**

Para ilustrar cómo funciona la paravirtualización en la práctica, el Código 3 presenta un ejemplo simplificado de cómo un sistema operativo paravirtualizado podría manejar operaciones de E/S.

// Código 3 - Ejemplo de operación paravirtualizada de E/S

#include <paravirt\_ops.h>

// Estructura para operaciones paravirtualizadas

struct paravirt\_ops pv\_ops = {

.read\_disk = hypercall\_read\_disk,

.write\_disk = hypercall\_write\_disk,

.alloc\_memory = hypercall\_alloc\_memory,

.free\_memory = hypercall\_free\_memory

};

// Función paravirtualizada para lectura de disco

int paravirt\_disk\_read(int device\_id, void\* buffer, size\_t size, off\_t offset) {

// En lugar de acceder directamente al hardware,

// realizamos una llamada al hipervisor

struct hypercall\_params params = {

.device\_id = device\_id,

.buffer = buffer,

.size = size,

.offset = offset,

.operation = HYPERCALL\_DISK\_READ

};

// Llamada directa al hipervisor

return pv\_ops.read\_disk(&params);

}

// Función tradicional vs paravirtualizada

void comparison\_example() {

// Método tradicional (emulado)

// outb(DISK\_COMMAND\_REG, READ\_SECTOR); // Instrucción sensible

// while(!(inb(DISK\_STATUS\_REG) & READY\_BIT)); // Polling

// Método paravirtualizado

char buffer[512];

int result = paravirt\_disk\_read(0, buffer, 512, 1024);

if (result == HYPERCALL\_SUCCESS) {

printf("Lectura exitosa mediante paravirtualización\n");

}

}

Código . Ejemplo de implementación de operaciones paravirtualizadas de E/S.

Este código ilustra la diferencia fundamental: donde un sistema tradicional utilizaría instrucciones de hardware directas (que requieren emulación), el sistema paravirtualizado utiliza llamadas específicas al hipervisor que son más eficientes y directas.

### **6.2.4) CONTENEDORES**

#### **INTRODUCCIÓN**

Los contenedores representan una revolución paradigmática en el mundo de la virtualización, ofreciendo una alternativa fundamentalmente diferente a los enfoques tradicionales de virtualización completa [Wolf et al., 2015]. Una estrategia completamente distinta para la creación de máquinas virtuales es la de contenedores [Wolf et al., 2015]. Esta tecnología ha transformado la manera en que desarrollamos, desplegamos y gestionamos aplicaciones, convirtiéndose en una piedra angular de la computación en la nube moderna y las arquitecturas de microservicios [Wolf et al., 2015].

A diferencia de las tecnologías de virtualización previamente discutidas, que buscan crear la ilusión de hardware completamente separado, los contenedores adoptan un enfoque radicalmente diferente: al emplear contenedores sólo se ejecuta un sistema operativo, que es el mismo para los sistemas anfitrión y huésped [Wolf et al., 2015]

#### **FUNDAMENTOS CONCEPTUALES Y EVOLUCIÓN HISTÓRICA**

**RAÍCES CONCEPTUALES: LA LLAMADA CHROOT()**

Los contenedores modernos tienen sus raíces conceptuales en una función aparentemente simple pero revolucionaria en su momento: la llamada al sistema chroot(). Esta función restringe la visión del sistema de archivos de un proceso a sólo el directorio hacia el cual ésta fue invocada [Wolf et al., 2015]. Aunque puede parecer una funcionalidad básica, chroot() estableció el principio fundamental de aislamiento que eventualmente evolucionaría hacia los sistemas de contenedores sofisticados que conocemos hoy.

La Tabla 14 presenta la evolución histórica de las tecnologías de contenedores, desde sus orígenes hasta las implementaciones modernas.

|  |  |  |  |
| --- | --- | --- | --- |
| **AÑO** | **TECNOLOGÍA** | **SISTEMA** | **INNOVACIÓN PRINCIPAL** |
| 1982 | chroot() | Unix 4.2BSD | Aislamiento básico del sistema de archivos |
| 2000 | FreeBSD Jails | FreeBSD 4.0 | Primer sistema completo de contenedores |
| 2001 | Linux VServer | Linux | Virtualización a nivel de SO en Linux |
| 2004 | Solaris Containers | Solaris 10 | Zones con gestión avanzada de recursos |
| 2006 | Process Containers | Linux | Precursor de cgroups |
| 2008 | LXC | Linux | Contenedores nativos de Linux |
| 2013 | Docker | Linux | Popularización y estandarización |
| 2014 | Kubernetes | Multiplataforma | Orquestación de contenedores |

Tabla . Evolución Histórica de las Tecnologías de Contenedores.

La historia de chroot() es particularmente interesante porque inicialmente se pensaba que había sido creada por Bill Joy en 1982 para ayudarse en el desarrollo del sistema Unix 4.2BSD. Sin embargo, investigaciones posteriores realizadas por Losh (2020) han demostrado que chroot() fue introducida en la 7ª Edición de Unix de Bell Labs en 1979, dos años antes de lo que se creía originalmente. Este descubrimiento surgió al examinar cuidadosamente los registros históricos del sistema SCCS (Source Code Control System).

**LA PRIMERA IMPLEMENTACIÓN COMPLETA: FREEBSD JAILS**

El salto conceptual de chroot() a un sistema completo de contenedores ocurrió cuando FreeBSD creó el subsistema Jails a partir de su versión 4.0, del año 2000 [Kamp, 2000; FreeBSD Foundation, 2022]. Esta implementación fue desarrollada por Poul-Henning Kamp en respuesta a una solicitud específica de Derrick T. Woolworth, quien necesitaba ejecutar múltiples versiones de Apache y MySQL en el mismo servidor para su empresa de hosting R&D Associates Inc [FreeBSD Wiki, 2025].

Esta implementación representó la primera realización completa de la visión de contenedores como la conocemos hoy, proporcionando no solo aislamiento del sistema de archivos, sino un entorno completamente aislado para la ejecución de aplicaciones.

#### **ARQUITECTURA Y DIFERENCIAS FUNDAMENTALES**

**COMPARACIÓN ARQUITECTÓNICA FUNDAMENTAL**

Mientras que las tecnologías antes descritas de virtualización implementan hardware virtual para cada sistema operativo, los contenedores más bien presentan un sistema operativo virtual para el conjunto de procesos que definen el comportamiento de cada máquina virtual [Wolf et al., 2015]. Mientras que los métodos tradicionales operan mediante una multiplexación de máquinas virtuales sobre hardware real, los contenedores operan mediante restricciones adicionales sobre los procesos de usuario [Wolf et al., 2015].

La Tabla 15 presenta una comparación arquitectónica detallada entre contenedores y otros métodos de virtualización.

|  |  |  |  |
| --- | --- | --- | --- |
| **ASPECTO** | **VIRTUALIZACIÓN COMPLETA** | **PARAVIRTUALIZACIÓN** | **CONTENEDORES** |
| Sistemas operativos | Múltiples SO completos | Múltiples SO modificados | Un SO compartido |
| Hipervisor | Requerido | Requerido (microkernel) | No requerido |
| Overhead de memoria | Alto (múltiples kernels) | Medio (kernels optimizados) | Bajo (kernel compartido) |
| Tiempo de inicio | Minutos | Minutos | Segundos |
| Aislamiento | Hardware virtual | API del hipervisor | Namespaces y cgroups |
| Flexibilidad de SO | Máxima | Alta | Limitada al SO anfitrión |
| Eficiencia de recursos | Baja | Media | Alta |

Tabla . Comparación Arquitectónica: Contenedores vs. Otros Métodos de Virtualización (elaboración propia basada en Wolf et al., 2015; Merkel, 2014).

Esta diferencia arquitectónica fundamental tiene implicaciones profundas. Mientras que los métodos tradicionales operan mediante una multiplexación de máquinas virtuales sobre hardware real, los contenedores operan mediante restricciones adicionales sobre los procesos de usuario [Wolf et al., 2015].

#### **MECANISMOS DE AISLAMIENTO EN CONTENEDORES**

**ARQUITECTURA DE AISLAMIENTO**

El aislamiento en contenedores se logra a través de múltiples mecanismos coordinados que trabajan en conjunto para crear la ilusión de sistemas separados [Wolf et al., 2015]. El núcleo del sistema crea un grupo para cada contenedor (también conocido como contexto de seguridad), aislándolos entre sí [Wolf et al., 2015] en varias áreas críticas.

La Figura 4 ilustra la arquitectura de aislamiento en contenedores, mostrando los diferentes mecanismos que trabajan en conjunto.

Interfaz de usuario gráfica, Aplicación

El contenido generado por IA puede ser incorrecto.

Figura . Arquitectura de Aislamiento en Contenedores.

#### **COMPONENTES ESPECÍFICOS DE AISLAMIENTO**

**AISLAMIENTO DE TABLAS DE PROCESOS**

Uno de los aspectos más fundamentales del aislamiento de contenedores es la gestión de procesos. Cada contenedor inicia su existencia ejecutando un init propio y enmascarando su identificador de proceso real por el número 1 [Wolf et al., 2015]. Este mecanismo crea la ilusión, para los procesos dentro del contenedor, de que están ejecutándose en un sistema completamente independiente.

**AISLAMIENTO DE COMUNICACIÓN ENTRE PROCESOS**

La seguridad y estabilidad del sistema requieren que ningún proceso de un contenedor debe poder interferir con la ejecución de uno en otro contenedor. Para lograr esto, el núcleo restringe toda comunicación entre procesos, regiones de memoria compartida y envío de señales entre procesos de distintos grupos [Wolf et al., 2015].

**AISLAMIENTO DE INTERFACES DE RED**

La conectividad de red en contenedores se maneja a través de interfaces virtuales separadas. Cada contenedor tendrá una interfaz de red con una dirección de acceso a medio (MAC) distinta [Wolf et al., 2015], permitiendo que cada contenedor tenga su propia identidad de red única y configuración de conectividad independiente.

**GESTIÓN DE DISPOSITIVOS DE HARDWARE**

El acceso al hardware en contenedores está cuidadosamente controlado. Normalmente los sistemas huésped no tienen acceso directo a ningún dispositivo en hardware [Wolf et al., 2015]. Esta restricción es fundamental para mantener la seguridad y estabilidad del sistema anfitrión.

**LÍMITES DE RECURSOS**

Una característica crucial de los contenedores modernos es su capacidad para gestionar recursos. Casi todas las implementaciones permiten asignar cotas máximas para el consumo de recursos compartidos, como espacio de memoria o disco o tiempo de CPU empleados por cada uno de los contenedores [Wolf et al., 2015].

La Tabla 16 describe los tipos de recursos gestionados en contenedores y sus impactos.

|  |  |  |  |
| --- | --- | --- | --- |
| **RECURSO** | **MÉTRICA** | **PROPÓSITO** | **IMPACTO DE LÍMITES** |
| CPU | Ciclos de procesador, % de uso | Prevenir monopolización | Throttling de procesos |
| Memoria | RAM en MB/GB | Evitar agotamiento del sistema | OOM killer activado |
| Disco | Espacio en MB/GB, IOPS | Controlar almacenamiento | Fallos de escritura |
| Red | Ancho de banda, paquetes/seg | Gestionar conectividad | Limiting de tráfico |
| Procesos | Número máximo de PIDs | Prevenir fork bombs | Fallos de creación |
| Archivos | Descriptores abiertos | Controlar recursos de SO | Error en apertura |

Tabla . Tipos de Recursos Gestionados en Contenedores (elaboración propia basada en Wolf et al., 2015; Merkel, 2014).

#### **VENTAJAS OPERACIONALES DE LOS CONTENEDORES**

**EFICIENCIA DE RECURSOS OPTIMIZADA**

Una de las ventajas más significativas de los contenedores es un consumo de recursos óptimo [Wolf et al., 2015]. Esta eficiencia contrasta marcadamente con otros métodos de virtualización donde una máquina virtual siempre ocupará algunos recursos, así esté inactiva [Wolf et al., 2015].

La diferencia es fundamental: mientras que las máquinas virtuales tradicionales mantienen un overhead constante debido a la necesidad de emular hardware y ejecutar sistemas operativos completos, en contenedores una máquina virtual que no tiene trabajo se convierte sencillamente en un grupo de procesos dormidos, probables candidatos a ser paginados a disco [Wolf et al., 2015].

La Tabla 17 presenta una comparativa de eficiencia entre contenedores y virtualización tradicional.

|  |  |  |  |
| --- | --- | --- | --- |
| **MÉTRICA** | **VIRTUALIZACIÓN TRADICIONAL** | **CONTENEDORES** | **MEJORA** |
| Tiempo de inicio | 30-60 segundos | 1-3 segundos | 90-95% |
| Memoria base | 512MB - 2GB | 1-50MB | 95-98% |
| Overhead de CPU | 5-15% | <1% | 85-95% |
| Densidad de instancias | 10-20 por servidor | 100-1000 por servidor | 500-5000% |
| Tiempo de despliegue | Minutos | Segundos | 90-95% |
| Tamaño de imagen | 1-10GB | 10-500MB | 80-95% |

Tabla . Comparativa de Eficiencia: Contenedores vs. Virtualización Tradicional (adaptado de Merkel, 2014; Felter et al., 2015; Sharma et al., 2016).

**VELOCIDAD DE DESPLIEGUE Y ESCALABILIDAD**

Los contenedores han revolucionado los tiempos de despliegue de aplicaciones [Wolf et al., 2015]. Donde las máquinas virtuales tradicionales requieren arrancar un sistema operativo completo, los contenedores pueden iniciar aplicaciones en cuestión de segundos. Esta capacidad es crucial para:

* Escalado automático: Respuesta rápida a cambios en la demanda
* Desarrollo ágil: Ciclos de desarrollo más rápidos
* Microservicios: Arquitecturas distribuidas eficientes
* CI/CD: Pipelines de integración y despliegue continuos

#### **LIMITACIONES Y CONSIDERACIONES**

**PÉRDIDA DE FLEXIBILIDAD ARQUITECTÓNICA**

Aunque los contenedores ofrecen numerosas ventajas, también presentan limitaciones importantes. Al operar a un nivel más alto, un contenedor presenta algunas limitantes adicionales (principalmente, se pierde la flexibilidad de ejecutar sistemas operativos distintos) [Wolf et al., 2015].

Esta limitación es particularmente relevante en entornos que requieren:

* Sistemas operativos heterogéneos
* Aplicaciones legacy con dependencias específicas de SO
* Entornos de testing que requieren múltiples sistemas operativos
* Aplicaciones con requisitos de kernel específicos

**FALTA DE UNIVERSALIDAD**

Otro desafío significativo es que se pierde buena parte de la universalidad mencionada en las secciones anteriores [Wolf et al., 2015]. Las diferentes implementaciones de contenedores comparten principios básicos de operación, pero la manera en que logran la separación e incluso la nomenclatura que emplean difieren fuertemente [Wolf et al., 2015].

La Tabla 18 muestra las variaciones en implementaciones de contenedores según la plataforma.

|  |  |  |  |
| --- | --- | --- | --- |
| **PLATAFORMA** | **TECNOLOGÍA BASE** | **NOMENCLATURA** | **CARACTERÍSTICAS DISTINTIVAS** |
| Linux | namespaces + cgroups | Containers | Implementación de referencia |
| FreeBSD | Jails | Jails | Primer sistema completo |
| Solaris | Zones | Zones | Gestión avanzada de recursos |
| Windows | Windows Containers | Containers | Compatibilidad con aplicaciones Windows |
| AIX | WPARs | partitions | Workload Partitions |

Tabla . Variaciones en Implementaciones de Contenedores (elaboración propia basada en Wolf et al., 2015; FreeBSD Wiki, 2025).

#### **EJEMPLO PRÁCTICO: IMPLEMENTACIÓN DE CONTENEDORES**

Para ilustrar los conceptos fundamentales de los contenedores, el Código 4 presenta un ejemplo práctico que demuestra la creación, configuración y gestión de un contenedor simple.

// Código 4 - Ejemplo simplificado de implementación de contenedores

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/wait.h>

#include <sys/mount.h>

#include <sched.h>

#include <signal.h>

#define STACK\_SIZE (1024 \* 1024) // 1MB stack

// Estructura para configuración del contenedor

struct container\_config {

char\* hostname;

char\* root\_path;

int cpu\_limit;

int memory\_limit;

char\* network\_interface;

};

// Función que se ejecutará dentro del contenedor

int container\_main(void\* arg) {

struct container\_config\* config = (struct container\_config\*)arg;

// 1. Cambiar el hostname del contenedor

if (sethostname(config->hostname, strlen(config->hostname)) == -1) {

perror("Error setting hostname");

return 1;

}

// 2. Cambiar root directory (chroot)

if (chroot(config->root\_path) == -1) {

perror("Error in chroot");

return 1;

}

// 3. Cambiar al directorio root del nuevo filesystem

if (chdir("/") == -1) {

perror("Error changing directory");

return 1;

}

// 4. Montar /proc para el contenedor

if (mount("proc", "/proc", "proc", 0, NULL) == -1) {

perror("Error mounting /proc");

return 1;

}

// 5. Inicializar el proceso principal del contenedor

printf("Contenedor iniciado correctamente!\n");

printf("Hostname: %s\n", config->hostname);

printf("Root path: %s\n", config->root\_path);

// 6. Ejecutar shell interactivo

execl("/bin/bash", "bash", NULL);

return 0;

}

// Función para crear un nuevo contenedor

int create\_container(struct container\_config\* config) {

// Alocar stack para el nuevo proceso

char\* stack = malloc(STACK\_SIZE);

if (!stack) {

perror("Error allocating stack");

return -1;

}

// Crear nuevo proceso con namespaces aislados

int flags = CLONE\_NEWPID | // Nuevo namespace de procesos

CLONE\_NEWNS | // Nuevo namespace de filesystems

CLONE\_NEWNET | // Nuevo namespace de red

CLONE\_NEWUTS | // Nuevo namespace de hostname

CLONE\_NEWIPC; // Nuevo namespace de IPC

pid\_t container\_pid = clone(container\_main,

stack + STACK\_SIZE,

flags | SIGCHLD,

config);

if (container\_pid == -1) {

perror("Error creating container");

free(stack);

return -1;

}

printf("Contenedor creado con PID: %d\n", container\_pid);

// Esperar a que termine el contenedor

int status;

waitpid(container\_pid, &status, 0);

free(stack);

return 0;

}

// Función principal

int main(int argc, char\* argv[]) {

// Verificar privilegios de root

if (getuid() != 0) {

fprintf(stderr, "Este programa requiere privilegios de root\n");

return 1;

}

// Configuración del contenedor

struct container\_config config = {

.hostname = "mi-contenedor",

.root\_path = "/tmp/container-root",

.cpu\_limit = 50, // 50% CPU

.memory\_limit = 512, // 512MB RAM

.network\_interface = "veth0"

};

printf("Iniciando contenedor...\n");

printf("Configuración:\n");

printf(" Hostname: %s\n", config.hostname);

printf(" Root path: %s\n", config.root\_path);

printf(" CPU limit: %d%%\n", config.cpu\_limit);

printf(" Memory limit: %dMB\n", config.memory\_limit);

// Crear y ejecutar el contenedor

if (create\_container(&config) == -1) {

fprintf(stderr, "Error creando contenedor\n");

return 1;

}

printf("Contenedor terminado exitosamente\n");

return 0;

}

Código . Ejemplo simplificado de implementación de contenedores usando namespaces de Linux.

Este código ilustra los conceptos fundamentales de la creación de contenedores:

* **Aislamiento de namespaces:** Uso de CLONE\_NEW\* flags para crear espacios de nombres separados
* **Cambio de root filesystem:** Implementación de chroot() para aislar el sistema de archivos
* **Configuración de hostname:** Personalización del entorno del contenedor
* **Montaje de sistemas de archivos:** Configuración del entorno /proc del contenedor

### **ACTIVIDAD DE APRENDIZAJE 2:** **VIRTUALIZACIÓN**

Lee cuidadosamente cada una de las oraciones incompletas relacionadas con el tema "Virtualización". Utiliza el banco de palabras clave proporcionado para completar correctamente los espacios en blanco.

Cada palabra del banco se usa una sola vez. Escribe la palabra que consideres correcta en el espacio correspondiente.

**PREGUNTAS**

1. La \_\_\_\_\_\_\_\_\_\_\_\_\_ virtual permite que los programas piensen que tienen acceso a un espacio de memoria exclusivo, incluso cuando la memoria física está siendo compartida.
2. La \_\_\_\_\_\_\_\_\_\_\_\_\_ (Unidad de Gestión de Memoria) se encarga de convertir las direcciones virtuales en direcciones físicas.
3. La \_\_\_\_\_\_\_\_\_\_\_\_\_ permite que un sistema ejecute software diseñado para una arquitectura completamente diferente mediante traducción de instrucciones.
4. La virtualización asistida por \_\_\_\_\_\_\_\_\_\_\_\_\_ utiliza extensiones específicas del procesador como Intel VT-x y AMD-V para mejorar el rendimiento.
5. En la \_\_\_\_\_\_\_\_\_\_\_\_\_, el sistema operativo huésped está consciente de que ejecuta en un entorno virtualizado y utiliza llamadas específicas al hipervisor.
6. Los \_\_\_\_\_\_\_\_\_\_\_\_\_ representan una alternativa a la virtualización completa, ya que comparten el mismo sistema operativo entre el anfitrión y los huéspedes.
7. El \_\_\_\_\_\_\_\_\_\_\_\_\_ es la capa de software que gestiona y coordina múltiples máquinas virtuales en un sistema.
8. La función \_\_\_\_\_\_\_\_\_\_\_\_\_ fue una de las primeras implementaciones que restringía la visión del sistema de archivos de un proceso a un directorio específico.
9. Los \_\_\_\_\_\_\_\_\_\_\_\_\_ en Linux proporcionan aislamiento de procesos, red, y otros recursos del sistema para crear contenedores.
10. El \_\_\_\_\_\_\_\_\_\_\_\_\_ computacional se refiere al costo adicional en rendimiento que introduce la virtualización comparado con la ejecución nativa.

**BANCO DE PALABRAS**

memoria

MMU

emulación

hardware

paravirtualización

contenedores

hipervisor

chroot

namespaces

overhead

**SOLUCIONES**

1. memoria
2. MMU
3. emulación
4. hardware
5. paravirtualización
6. contenedores
7. hipervisor
8. chroot
9. namespaces
10. overhead

# **REFERENCIAS**

Amsden, Z., Arai, R., Hecht, D., Holler, A., & Subrahmanyam, P. (2006). VMI: An interface for paravirtualization. En Ottawa Linux Symposium (pp. 371-386).

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., & Warfield, A. (2003). Xen and the art of virtualization. ACM SIGOPS Operating Systems Review, 37(5), 164-177.

Carretero Pérez, J., García Carballeira, F., De Miguel Anasagasti, P., & Pérez Costoya, F. (2001). Sistemas operativos: Una visión aplicada. McGraw-Hill.

Felter, W., Ferreira, A., Rajamony, R., & Rubio, J. (2015). An updated performance comparison of virtual machines and Linux containers (Reporte técnico). IBM Research.

FreeBSD Foundation. (2022). An introduction to FreeBSD jails. <https://freebsdfoundation.org/freebsd-project/resources/introduction-to-freebsd-jails/>

Kamp, P. H. (2000). Jails: Confining the omnipotent root. En Proceedings of the 2nd International SANE Conference. FreeBSD Project.

Kivity, A., Kamay, Y., Laor, D., Lublin, U., & Liguori, A. (2007). KVM: The Linux virtual machine monitor. En Ottawa Linux Symposium (pp. 225-230).

Merkel, D. (2014). Docker: Lightweight Linux containers for consistent development and deployment. Linux Journal, 2014(239), 2.

Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). Operating system concepts (10.ª ed.). John Wiley & Sons.

Stallings, W. (2018). Operating systems: Internals and design principles (9.ª ed.). Pearson.

Tanenbaum, A. S., & Bos, H. (2015). Modern operating systems (4.ª ed.). Pearson.

Wolf, G., Ruiz Baun, C., & Schmid, E. (2015). Sistemas operativos: Una visión aplicada. Alfaomega Grupo Editor.