- Introducción
  - Hardware Vs Software (SBZ-1.2.1)
  - Historia - Sistemas Batch
    - Operadores
    - Sistemas Operativos
- El sistema operativo
  - Organización de la Computadora
    - Distribución básica de una computadora:
    - Arranque de una PC (Unix-like) (SBZ-1.2.1)
    - Almacenamiento (SBZ-1.2.2)
      - Memoria principal
      - Memoria Secundaria
      - Estructura de IO
  - Arquitectura de la Computadora
    - Sistemas Monoprocesador
    - Sistemas Multiprocesador
      - Multiprocesamiento Asimétrico (AMP)
      - Multiprocesamiento Simétrico (SMP)
      - Acceso a memoria uniforme (UMA)
      - Acceso a memoria no-uniforme (NUMA)
    - Sistemas Cluster
- Procesos
  - Definiciones
    - Programa (estático):
    - Proceso (dinámico):
  - Process Control Block (PCB)
  - Máquina de estados
    - Estados
  - Acciones de un proceso
  - Acciones del SO
    - Scheduling
    - Context Switch
      - Context Switch: Artículos para leer
  - Arbol de Procesos
- API
  - Llamadas al Sistema (SYSCALLS)
    - Tipos de llamadas al sistema
    - POSIX
- Scheduling
  - Políticas de Scheduling
  - Objetivos
    - Soluciones de Compromiso
  - Cooperativo vs Con Desalojo.
    - Con Desalojo (Preemptive, Apropiativo)
    - Cooperativo
  - Tipos de Scheduling
    - FIFO / FCFS
    - Round-Robin
      - ¿Cuánto quantum es adecuado?
      - Prioridades
    - Múltiples colas

        - SJF
        - Real-Time
        - SMP
    - Consideraciones Prácticas
        - Consideraciones Básicas
        - Consideraciones Adicionales
- Sincronización entre Procesos
    - Condición de carrera (Race Condition)
    - Mecanismos de Sincronización
        - Sección Crítica
        - Test-And-Set
        - TAS-Lock (Test-And-Set Lock) (spin lock)
            - Problema: Busy Waiting
            - Soluciones al Busy Waiting
        - TTAS-Lock (Test and Test-And-Set Lock) (local spinning)
        - Otros objetos atómicos
        - Semáforo
        - Mutex
    - Modelo Coffmann (para detectar Deadlock)
    - Modelo de Grafos Bipartitos (para detectar Deadlock)
    - Problemas de sincronización
        - Prevención
        - Detección
    - Modelo teórico
        - Propiedades
            - Exclusión Mutua (EXCL)
            - Progreso (PROG)
            - Progreso global absoluto (WAIT-FREE)
            - Progreso global dependiente (G-PROG)
            - Justicia (FAIR) (fairness)
            - Observaciones
    - Bibliografía
- Sincronización
    - Contención
    - Race Condition / Condición de carrera
        - Ejemplo de Race Condition:
    - Critical Section / Sección Crítica
        - Resolución de Sección Crítica
    - Algoritmo de Peterson
        - Peterson - idea
        - Peterson - correctitud
        - Peterson - Desventajas
    - Primitivas de Hardware
    - Solución genérica a sección crítica usando locks
    - Test-And-Set
        - Test-And-Set: Shared Boolean variable lock
        - Test-And-Set: Bounded-waiting Mutual Exclusion
    - Compare-And-Swap
        - Compare-And-Swap: Shared integer “lock”
    - Get-And-Inc
    - Get-And-Add

    - Spinlock / Test-And-Set lock / TASLock
        - Usando primitivas del Sistemas Operativo
        - Usando TAS
    - Local Spinning / TTASLock
    - Cola Atómica
    - Mutex Recursivo
    - Deadlock
        - Modelo Coffmann (para detectar Deadlock)
        - Modelo de Grafos Bipartitos (para detectar Deadlock)
    - Registros Atómicos
        - Exclusión Mútua
        - Consenso
- Administración de Memoria
    - Roles de un subsistema de memoria
    - Objetivos
    - Problemas
    - Manejo del espacio libre / Fragmentación
        - Fragmentación
            - Fragmentación Externa
            - Fragmentación Interna
        - Organización de la Memoria
            - Mapa de Memoria de una Tarea
            - Bitmap
            - Lista Enlazada
        - Memoria Virtual
- Entrada / Salida
    - Tipos de dispositivos
    - Dispositivos de Almacenamiento
    - Esquema de un dispositivo (en capas)
        - Desde el Hardware
        - Desde el Software
    - Drivers
        - Formas de interactuar con el Hardware
            - Polling
            - Interrupciones (o push)
            - DMA
    - Subsistema de E/S
        - Char Device
        - Block Device
    - Subsistema de E/S Linux:
        - Características:
        - API del subsistema de E/S de Linux:
    - Planificación de E/S:
        - Políticas de scheduling de E/S a disco:
    - SSD: Solid State Drive
    - Spooling
    - Otros usos: Locking
    - Protección de la información
        - Copias de Seguridad (backup)
        - Redundancia
            - Métodos:
            - Peligros de RAID

- Sistemas de Archivos
  - Archivo:
    - Atributos de un Archivo:
    - Operaciones de un Archivo:
  - Filesystems:
  - Filesystem populares:
  - Responsabilidades de un Filesystem:
    - Organización lógica de los archivos (visible desde fuera).
    - Nombre y ruta de los archivos.
    - Punto de montaje.
    - Representación del archivo (¡Importante!)
    - Gestión del espacio libre.
    - Metadatos.
  - Representación de archivos
    - Solución de DOS, FAT:
    - Solución de Unix, Inodos:
      - Inodos: Directorios
      - Inodos: Links
  - Atributos / Metadatos de un filesystem
  - Manejo del espacio libre
  - Caché
  - Consistencia
  - Características Avanzadas
    - Cuotas de disco
    - Encriptación
    - Snapshots
    - Raid por Software
    - Compresión
  - Performance de un filesystem
  - NFS: Network File System
  - EXT2
- Sistemas Distribuídos
  - Ejemplos:
  - Ventajas:
  - Problemas:
  - Sistemas con Memoria compartida
    - Por Hardware
    - Por Software
  - Sistemas sin Memoria Compartida
    - Sincrónicos
    - Asincrónicos
      - Pasaje de Mensajes
  - Locks en Sistemas Distribuídos
    - Centralizado
  - Lamport
    - Idea general
    - Implementación:
  - Acuerdo Bizantino
    - Teoremas
  - Clusters
  - Scheduling en Sistemas Distribuídos

        - Factores de una Política de Scheduling
- Protección y Seguridad
    - Seguridad de la Información
    - Características de un Sistema de Seguridad
    - Propiedades de un Sistema de Seguridad
    - Criptografía
        - Algoritmos de Encriptación Simétricos
        - Algoritmos de Encriptación Asimétricos
        - Algoritmos de Hash One-Way
    - Funciones de Hash
    - Método RSA
        - Explicación RSA
        - Firma digital con RSA
        - Autenticación remota con hash
    - Autorización: Permisos
        - DAC vs MAC
        - DAC en Unix
        - Puntos importantes
- Sistemas Distribuidos
    - Modelo de Fallas
    - Métricas
    - Problemas
    - Exclusión mutua distribuida
    - Locks Distribuídos
    - Elección de Líder
    - Instantánea Global Consistente
    - Two-Phase Commit
    - Consenso: acuerdos y aplicaciones
    - Bibliografía Adicional
- Microkernels
    - Ventajas vs Kernel Monolítico
    - Idea Inicial
    - En la práctica...
    - Ideas útiles
- Virtualización
    - Conceptos:
    - Simulación
        - Problemas:
    - Emulación de Hardware
        - Problemas:
    - Virtualización asistida por Hardware
    - Desafíos / Problemas
    - Escenarios de uso / Ventajas
    - Bibliografía

Introducción

Hardware Vs Software (SBZ-1.2.1)

División de los sistemas informáticos entre **Hardware < - > Software**. El sistema operativo es el **intermediario** entre ellos:

- **Visión del usuario**: Para que el software no se tenga que preocupar de detalles de bajo nivel. Los distintos contextos de uso van a requerir distintos objetivos a cumplir por el sistema operativo.
    - Computadoras de escritorio de un solo usuario: se prioriza la facilidad de uso, seguido de la performance, pero no se le da prioridad a la utilización de recursos.
    - Workstations que pertenecen a una red: tienen recursos dedicados indivualmente, se intenta buscar un equilibrio entre la usabilidad y el acceso a los recursos locales vs el acceso compartido a los recursos de red.
    - Terminales conectadas a un Mainframe: los usuarios no tienen recursos individuales, por lo que se intenta maximizar la utilización de recursos (uso eficiente de CPU, memoria, IO, etc).
- **Visión del fabricante**: Para que el hardware no se vea afectado por un mal uso por parte del software. El sistema operativo cumpliría las siguientes funciones:
    - Administrador de recursos (CPU, memoria, IO).
    - Resolución de problemas de contención.
    - Controlador de dispositivos y programas.

Historia - Sistemas Batch

- Antes las computadoras costaban millones, y estaban en ambientes dedicados.
- Se utilizaban tarjetas perforadas y lectores/impresoras para ejecutar programas.
- **Problema: mucho tiempo de procesamiento desperdiciado.**

Operadores

- Se agregaron computadoras intermediarias, que leían las tarjetas y grababan en cintas (+ rápido).
- Nació el rol del "operador", intermediario que debía encargarse de cargar los programas.
- Precursor del sistema operativo.

Sistemas Operativos

Las siguientes generaciones de computadoras ya tenían un SO formal. Esto solucionaba los problemas de desperdicio de recursos cuando el procesador se encontraba ocioso leyendo las cintas de memoria (mientras tanto, se procesaban otros trabajos).

- **Multiprogramación**. Correr muchos programas concurrentemente. El rendimiento aumenta, ya que si bien cada trabajo tarda lo mismo individualmente, la ejecución de los

trabajos *j1* + *j2* tarda menos que si no hubiera sistema operativo.

- **Contención**. Varios programas pueden querer acceder al mismo recurso a la vez.
- **Timesharing**. Conectar muchas terminales a una misma computadora, y darles un poco de tiempo de procesador a cada una.

El sistema operativo

Un sistema operativo hace de intermediario entre el hardware y los programas del usuario.

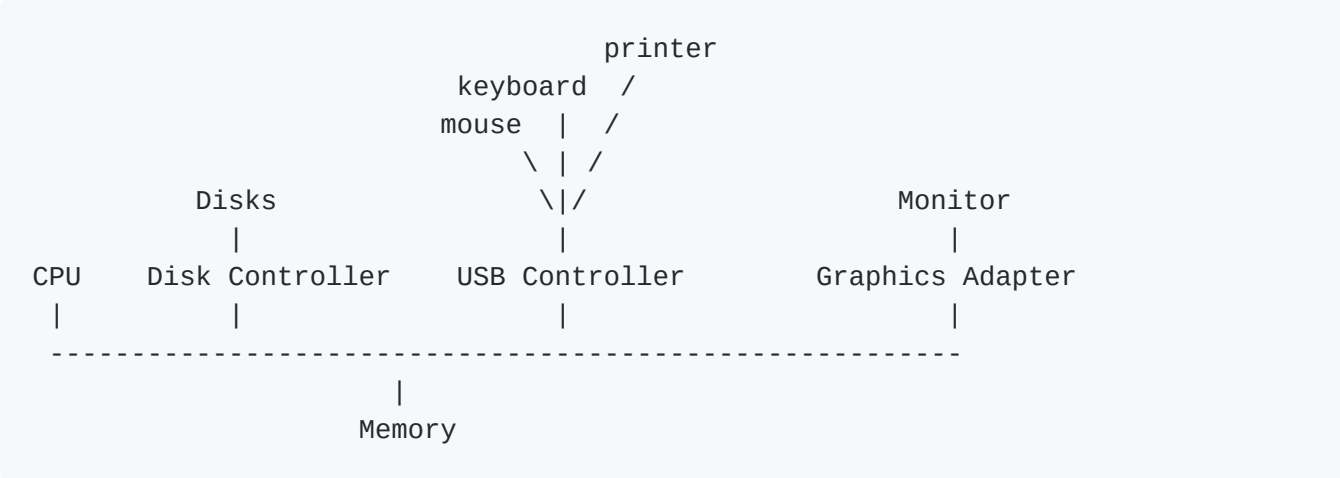
Tiene que manejar la contención y la concurrencia de forma de buscar un **balance óptimo entre el rendimiento y la correctitud**.

Al ser el programa más importante, corre con máximos privilegios (nivel 0), entendiendo al sistema operativo como el Kernel (programa más importante).

También, según la definición, se pueden llegar a considerar los programas de sistema (ejemplo, programas de configuración, drivers, etc), e incluso a veces los programas de aplicación (ej, editores, compiladores, etc) como parte integral del Sistema Operativo.

Organización de la Computadora

Distribución básica de una computadora:



Arranque de una PC (Unix-like) (SBZ-1.2.1)

1. Bootstrap Program: Programa muy pequeño alojado en la ROM (EEPROM/Firmware), que inicializa el sistema (CPU, dispositivos, memoria) y se encarga de buscar y cargar el sistema operativo (Kernel) en memoria.
2. Kernel: Programa que provee servicios a los demás programas (ya sean de sistema o de usuario).

    - En ocasiones, algunos de estos servicios son provistos por programas por fuera del propio Kernel (ejemplo daemons, servicios de sistema), o por programas que se incrustan como parte del Kernel (ejemplo Drivers).

3. Se carga el primer servicio/proceso de sistema (init), y este se encarga de cargar otros programas (ejemplo daemons).
    - A partir de este punto, el sistema operativo queda en estado de "esperando que sucedan eventos".

4. Cada vez que exista una interrupción o un system call (syscall, monitor call), esto es un evento que el sistema operativo "atiende".
    - Las interrupciones pueden ser l
    - La forma de atender estas interrupciones es mediante una función global de atención de interrupciones (lento), o mediante una tabla que describa las distintas rutinas de atención de interrupciones (interrupt vector). Existe una cantidad limitada de interrupciones (IRQs), y cada dispositivo se conecta unívocamente a una de ellas.
    - Cuando un sistema operativo atiende una interrupción, el estado actual de la CPU (ejemplo el Instruction Pointer, y en ocasiones los flags, los registros) es guardado (generalmente en la pila), y al terminar la interrupción todo debe ser restaurado a su estado original.

Almacenamiento (SBZ-1.2.2)

Los medios de almacenamiento se pueden organizar en una jerarquía:

```
Registros
Cache
RAM
Discos SSD
Discos magnéticos
Discos ópticos
Cintas magnéticas
```

En donde normalmente se cumple que los elementos que se encuentran más arriba, tienen mayor velocidad, pero son más caros y tienen menor capacidad.

Memoria principal

La CPU sólo puede correr instrucciones o cargar datos desde la memoria principal (RAM), así que cualquier programa que se desee correr debe ser previamente cargado allí.

Cualquier tipo de memoria puede ser pensado como un **array de bytes**, en donde **cada byte tiene su propia dirección**, y toda interacción puede ser vista como una **secuencia de LOAD y STORE**.

Idealmente, todos los programas deberían estar almacenados en memoria principal de forma permanente. Esto no es posible porque esta es relativamente pequeña, y por otro lado es volátil, por lo que al apagar el equipo toda la información contenida en la misma se pierde.

Memoria Secundaria

La memoria secundaria funciona como una extensión de la memoria principal, que es capaz de almacenar grandes cantidades de datos, y de retenerlos cuando esta se encuentra apagada.

Tipos de memoria secundaria:

- Discos rígidos (magnéticos, ssd, híbridos)
- Discos ópticos (CD-ROM, Blue-Ray)
- Memorias flash (pendrives, memorias SD)
- Otros (diskettes, cintas magnéticas)

Se diferencian según distintas características: capacidad, tamaño físico, precio, confiabilidad, durabilidad, velocidad, etc. Generalmente, siempre hay un tradeoff (mejorar una de las características tienen un impacto en alguna de las otras, por ejemplo, si aumentamos la capacidad aumenta el precio y/o el tamaño, si aumentamos la capacidad manteniendo el precio y el tamaño, disminuye la velocidad, o la confiabilidad, etc).

Estructura de IO

El almacenamiento se puede ver como uno de los tantos dispositivos de IO del sistema.

Cada tipo de almacenamiento está ligado a un controlador de dispositivo (ejemplo, los discos IDE están conectados a un controlador IDE, y los discos SCSI están conectados a un controlador SCSI). El controlador de dispositivo es un chip o placa independiente del procesador, y se encarga de hacer de intermediario entre los dispositivos y el procesador. Así, el controlador de dispositivo provee el acceso a los dispositivos que controla, mediante una interfaz única, que evita que el procesador tenga que encargarse de detalles de Hardware específicos de cada fabricante (ejemplo: mover los platillos de un disco).

Los Sistemas Operativos tienen un driver específico para cada tipo de controlador de dispositivo, que a su vez se encargan de hacer de intermediario entre el controlador y los programas (de usuario o de sistema) que hagan uso de estos dispositivos. Así, se provee de una interfaz uniforme para el acceso a cada tipo de controlador de dispositivo.

Normalmente, existen dos formas de acceder a los dispositivos. La primera, es seteando determinados parámetros (ejemplo desde donde se desea leer) en el controlador de dispositivos, y esperando que este a su vez se encargue de realizar la lectura correspondiente en el dispositivo, y luego devolver el resultado al procesador; esto es muy lento, y sólamente sirve para transferir pequeñas cantidades de datos. La segunda forma es mediante DMA, en donde después de que el procesador le setea los parámetros correspondientes, el controlador

de dispositivos se encarga de realizar la transferencia de datos directo desde el dispositivo hasta la memoria principal, sin la intervención del procesador; esto obviamente hace que el procesador pueda realizar otras tareas mientras se espera la transferencia. En ambos casos, el final de la transferencia se suele informar mediante una interrupción.

Arquitectura de la Computadora

Sistemas Monoprocesador

- Un solo procesador por sistema.
- Pueden haber otros procesadores para usos especificos (ejemplo, controlador de dispositivo, controlador de bus)
  - A veces estos son manejados por el propio sistema operativo a traves de mensajes.
  - Otras veces, el sistema operativo ni siquiera se puede comunicar con estos procesadores.
  - Sea como sea, el trabajo se realiza de forma autónoma por los otros procesadores.

Sistemas Multiprocesador

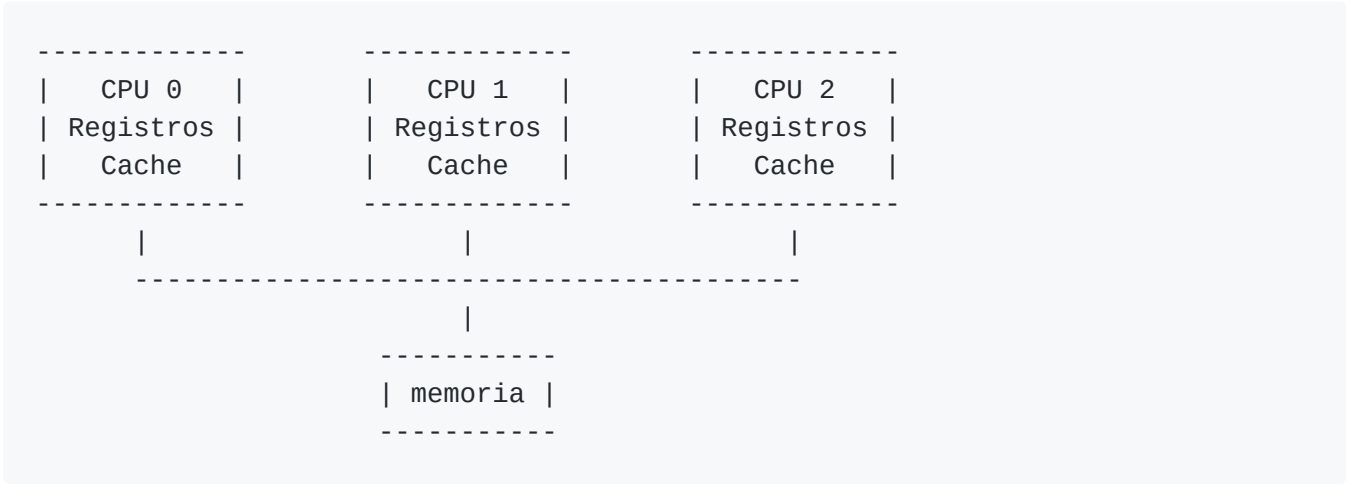
- También conocidos como sistemas paralelos o multinúcleo.
- Más de un procesador por sistema.
  - Funcionando al mismo tiempo.
  - Comunicándose entre sí.
  - Compartiendo recursos como el bus, el clock, la memoria.
- Mayor rendimiento: en el caso ideal se espera que al contar con N procesadores, el rendimiento sea N veces el de un solo procesador. En la práctica, esto no sucede, ya que contar con más de un procesador agrega un overhead, surgido del trabajo adicional de sincronización, y la contención generada por los recursos compartidos.
- Más barato: al aumentar la cantidad de procesadores, y compartir otros recursos (memoria, dispositivos, etc), se abarata el costo respecto a si se quisiera contar con un sistema de poder de cómputo similar, pero en donde se hubieran tenido que duplicar todos los recursos (ie. "varias PC completas interconectadas").
- Mayor confiabilidad / tolerancia a fallas: si se cae un procesador en un sistema multiprocesador, el resto de los procesadores pueden hacerse cargo del trabajo, lo cual disminuye el rendimiento, pero no hace caer el sistema. Esto se llama **graceful degradation**.

Multiprocesamiento Asimétrico (AMP)

- Hay un procesador jefe, que va designando tareas al resto de los procesadores.
- Los procesadores no tienen que ser necesariamente iguales.

Multiprocesamiento Simétrico (SMP)

- Todos los procesadores son iguales, y realizan las tareas controlados por el mismo sistema operativo.
- Se consideran todos pares.



Acceso a memoria uniforme (UMA)

- El acceso a memoria es igual desde cualquiera de los procesadores. (ver dibujo anterior)

Acceso a memoria no-uniforme (NUMA)

- Los procesadores pueden tener cada uno una determinada parte de la memoria en donde cuenten con acceso privilegiado.

Sistemas Cluster

- Se juntan varios nodos.
- Cada nodo es un sistema individualmente.
- Los nodos se comunican mediante LAN.

Procesos

Definiciones

Programa (estático):

- Texto escrito en un lenguaje de programación.
- Código compilado (objeto / máquina).

Proceso (dinámico):

- Tiene un estado.
  - Program Counter: instrucción actual.
  - Memoria Dinámica (heap).
  - Pila de Ejecución (stack).
- Tiene un identificador único **(PID)**.

Process Control Block (PCB)

Estructura que contiene la información del contexto proceso.

- Estado
- Program Counter, Stack Pointer
- Registros de la CPU
- Metadatos del scheduler (ejemplo prioridad)
- Metadatos del manejador de memoria, IO.

Máquina de estados

Estados

- **RUNNING**: Está usando la CPU.
- **WAITING**: Está bloqueado, usando o esperando algún recurso de IO.

- **READY**: Está libre, listo para correr.

Acciones de un proceso

- Realizar operaciones entre registro y memoria del usuario (User Address Space)
- Acceder a un servicio del kernel **(SYSCALL)**
  - Realizar IO
  - Lanzar un proceso hijo
  - Salir - `exit()`
    - Liberar todos los recursos del proceso.
    - Status de terminación.
    - Linux (C standard): EXIT_SUCCESS , EXIT_FAILURE
    - Este código de status le es reportado al padre.
- **Domain Crossing** a través de las System Calls

Acciones del SO

- Admitir un proceso **(NEW-->READY)**
- Otorgar (dispatch) CPU a un proceso **(READY-->RUNNING)**
- Desalojar un proceso:
  - Preemption: El proceso es interrumpido **(RUNNING->READY)**
  - Blocking: El proceso se bloquea al requerir un recurso **(RUNNING->WAITING)**

Scheduling

- ¿Cuánto tiempo se ejecuta el proceso?
  - Hasta que termine (sin time-sharing).
  - Un quantum.
- Los sistemas operativos modernos hacen **preemption**.
  - Scheduler: decide cómo y a quién le toca.
  - Context Switch: se cambia el contexto a otro proceso.

Context Switch

- Guardar el PCB del proceso desalojado.
- Cargar el PCB del proceso alojado.
- El tiempo usado en el cambio se desperdicia.
  - Duración: depende del Hardware
  - Cantidad: debe haber un Quantum apropiado.

- Se implementa a través de la interrupción de clock.

Context Switch: Artículos para leer

- Costo
  - Chuanpeng Li, Chen Ding, and Kai Shen. 2007. Quantifying the cost of context switch. http://goo.gl/GqGvKt
  - Francis M. David, et al. 2007. Context switch overheads for Linux on ARM platforms. http://goo.gl/pj9Dwj
- Prevención
  - Jaaskelainen, et al. Reducing context switch overhead with compiler-assisted threading. 2008. http://goo.gl/8th0dy.
  - Kloukinas, Yovine. 2011. A model-based approach for multiple QoS in scheduling: from models to implementation. http://goo.gl/4xL1JT

Arbol de Procesos

- Jerarquía.
- Existe un proceso **root**, o **init**.
- Los procesos pueden lanzar procesos hijos.
  - `fork()` crea un proceso igual al padre, y devuelve su PID.
  - `exec()` carga un binario ejecutable sobre un proceso existente.
  - `wait()` suspende al padre hasta que el hijo termine; cuando termina, recibe su status.

```
(parent) --> FORK() --> --> (parent) ------------------> WAIT() --> (resumes)
                        |                                  /
                        \--> child --> EXEC() --> EXIT() -->/
```

API

Llamadas al Sistema (SYSCALLS)

Tipos de llamadas al sistema

Tipo de llamada	Windows	Unix
Control de procesos	CreateProcess ExitProcess() WaitForSingleObject()	fork() exit() wait()

Tipo de llamada	Windows	Unix
Administración de archivos	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Administración de dispositivos	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Mantenimiento de información	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Comunicaciones	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()

POSIX

Portable Operating System Interface. And X from Unix.

- IEEE 1003.1/2008 http://goo.gl/k7WGnP
- POSIX.1: **Core Services**
  - Creación y control de procesos
  - Pipes
  - Señales
  - Operaciones de archivos y directorios
  - Excepciones
  - Errores del bus.
  - Biblioteca C
  - Instrucciones de E/S y de control de dispositivo (ioctl).
- POSIX.1b: **extensiones para tiempo real**
  - Planificación (scheduling) con prioridad.
  - Señales de tiempo real.
  - Temporizadores.
  - Semáforos.
  - Intercambio de mensajes (message passing).
  - Memoria compartida.
  - Entrada/salida síncrona y asíncrona.
  - Bloqueos de memoria.

- POSIX.1c: **extensiones para hilos (threads)**:
  - Creación, control y limpieza de hilos.
  - Planificación (scheduling).
  - Sincronización.
  - Manejo de señales.
- POSIX.2: **Shell y Utilidades**
  - Intérprete de Comandos
  - Programas de Utilidad

Scheduling

Políticas de Scheduling

- Parte fundamental de un Sistema Operativo.
- Muy importante optimizar.

Objetivos

- Ecuanimidad / Justicia: recibir **dosis "justas"** de CPU.
- Eficiencia: que la **CPU no desperdicie** recursos.
- Carga del sistema: **cantidad** de procesos en espera.
- Tiempo de espera: **tiempo** que un proceso está en espera.
- Latencia: **tiempo de respuesta** (resultados, interactividad)
- Tiempo de ejecución (*Completion Time* o *Turn Around*): tiempo en que cada proceso **termina**
- Rendimiento (*Throughput*): número de **procesos terminados** por tiempo
- Liberación de Recursos: **liberar** rápidamente los recursos.

Soluciones de Compromiso

> Cada política va a intentar **maximizar** uno o más **objetivos**, y **minimizar el impacto** en el resto.

- **No se puede tener todo**.
- Objetivos contradictorios.
- Distintos intereses <- depende de los usuarios ->

Cooperativo vs Con Desalojo.

Los Schedulers normalmente combinan ambos métodos.

Con Desalojo (Preemptive, Apropiativo)

> El scheduler utiliza la **interrupción de clock**, y decide si el proceso debe seguir ejecutándose.

- ¡Se requiere un **procesador con interrupciones** de clock!.
- El clock interrumpe sólamente **50 o 60 veces por segundo**.
- No se les da **garantías de continuidad** a los procesos
  - Esto es problema en REAL-TIME.

Cooperativo

> El scheduler analiza la situación **cuando el Kernel toma control** (en los syscalls).

- Entrada/Salida.
- Syscalls explícitos para permitir ejecución de otros procesos.

Tipos de Scheduling

FIFO / FCFS

First In - First Out / First Came - First Served

> Cualquier esquema de **prioridad fija** representa un **riesgo de inanición**.

- Se asume que *todos los procesos son iguales*.
- **Un solo proceso** grande puede **saturar la CPU**.
- Solución, **agregar prioridades**:
  - Problema: Inanición (Starvation): Los procesos de mayor prioridad **demoran infinitamente** a los de menor prioridad, que nunca se ejecutan.
- Solución 2, **aumentar prioridad con envejecimiento**.

Round-Robin

> Darle un poco de *quantum* a cada proceso, e ir alternando

¿Cuánto quantum es adecuado?

- Si es muy largo, los procesos interactivos podrían hacer parecer que el sistema no está respondiendo.
- Si es muy corto, el tiempo de cambio de contexto (desperdiciado) pasa a ser muy grande en relación al quantum, por lo que el SO pasa mucho tiempo haciendo "mantenimiento".

Prioridades

- Asignadas por el usuario (administrativas)
- Decididas por el proceso (no suele funcionar)
- A medida que el proceso envejece, la prioridad disminuye para evitar la inanición.
- Los procesos que hacen E/S reciben prioridad, al liberar su quantum antes de tiempo.

Múltiples colas

> Colas con distinto quantum

- menos quantum <=> más prioridad
- cuando un proceso agota su quantum, es promovido a una cola de más quantum
- procesos interactivos <- colas de mayor prioridad >
- cuando un proceso termina de hacer E/S, se lo vuelve a una cola de máxima prioridad
- la idea general es minimizar el tiempo de respuesta para procesos interactivos, asumiendo que los procesos largos son menos sensibles a demoras.

SJF

Shortest Job first

> Los procesos más cortos van primero

- Orientado a esquemas de trabajo batch.
- Se debe poder predecir el tiempo de duración de un proceso, o al menos clasificarlo.
- Si las predicciones son buenas, es óptimo en cuanto a latencia promedio.
- Una variante es no pensar en la duración total, sino en el que menos tiempo tenga hasta la próxima E/S.
    - El problema es saber cuánta CPU va a utilizar un proceso.
    - Una opción es usar información del pasado para predecir.
    - Si el proceso es irregular, no es posible.

Real-Time

> Los deadlines (tiempo de finalización) son estrictos

- Scheduling en RT es todo un mundo aparte, fuera del scope de la materia.
- Normalmente se entiende que el deadline es algo crítico: si no se lo cumple, algo malo va a pasar.
- Una opción es ejecutar el proceso que tenga un deadline más próximo.

SMP

Symmetric Multiprocessing

> Múltiples procesadores corriendo al mismo tiempo

- Como sucede con RT, es un problema en sí mismo.
- El caché es un problema.
  - Es de vital importancia para el rendimiento de los programas.
  - Si el scheduler hace pasar un proceso de un procesador a otro, se pierde la caché.
- Surge el concepto de **afinidad al procesador**.
  - Se divide en afinidad débil y afinidad fuerte.

Consideraciones Prácticas

Elegir un buen algoritmo de scheduling es difícil

Consideraciones Básicas

- ¿La ecuanimidad, es por proceso o por usuario? ¿Y un proceso con muchos hijos?
- ¿Qué hacer con un proceso que requiera mucha CPU?
- Un buen algoritmo de scheduling debe ajustarse a medida que cambien los patrones de uso.
  - A veces se pueden armar modelos matemáticos.
  - Se pueden tomar patrones de carga de sistemas estandarizados o benchmarks previos.
- Se puede obtener información útil a partir del comportamiento de un proceso.
  - Si un proceso abre una terminal, puede estar por volverse interactivo.
  - Se puede hacer un análisis estático.

Consideraciones Adicionales

- Usos específicos:
  - Bases de Datos.
  - Cómputo Científico.
  - Benchmarking.
- Threads
- Virtualización.

Ver gráfico en página 19

Sincronización entre Procesos

Ver qué pasa cuando se tienen ejecuciones Secuenciales vs Concurrentes.

¿Da lo mismo la forma en que se ejecutan los procesos?

- contención: dos procesos quieren acceder al mismo recurso
- concurrencia: dos procesos se ejecutan de forma simultánea

Condición de carrera (Race Condition)

> Fenómeno en donde los resultados varían según en qué orden se ejecuten las cosas.

Solución: Garantizar Exclusión Mutua mediante secciones críticas (CRIT).

Mecanismos de Sincronización

Sección Crítica

Es un pedazo de código tal que se cumple

1. sólo hay un proceso en CRIT
2. todo proceso esperando para CRIT va a entrar
3. ningún proceso fuera de CRIT puede bloquear a otro

Un proceso puede:

- entrar a la sección crítica
- salir de la sección crítica

Generalmente, se resuelve mediante locks (booleanos indicando si la sección está bloqueada) y es necesario utilizar hardware adicional.

Test-And-Set

> Se lee la variable, y al mismo tiempo se la setea en true. Este es un mecanismo provisto por el procesador, de forma que resulta atómico/indivisible y libre de bloqueos.

```
private atomic <bool> reg;
atomic bool get () {return reg;}
atomic void set(bool b) {reg = b;}
atomic bool getAndSet (bool b) {
    bool m = reg;
    reg = b;
    return m;
}
atomic bool testAndSet () {
```

```
    return getAndSet (true);
}
```

TAS-Lock (Test-And-Set Lock) (spin lock)

> Se utiliza la variable como un lock, y se hace TAS mediante Busy Waiting hasta que el valor leído sea falso (o sea, que se libere el lock).

- **no es atómico**
- la espera no es acotada

```
public class TASLock {
    private atomic <bool > reg;
    public void create () {
        reg.set(false);
    }
    public void lock () {
        while (reg.testAndSet()) {}
    }
    public void unlock () {
        reg.set(false);
    }
}
```

Problema: Busy Waiting

> El código se la pasa intentando obtener el lock de una forma agresiva.

- consume muchísima CPU
- **perjudica al resto de los procesos**
- Su overhead, sin embargo, puede ser menor que el de usar semáforos.

Soluciones al Busy Waiting

- Poner un sleep() en el cuerpo del while
  - poco tiempo -> desperdicia CPU
  - mucho tiempo -> mucha espera

```
void lock (time delay) {
    while (reg.testAndSet()) {sleep(delay);}
}
```

- No iterar sobre TestAndSet(), es decir, testear antes de intentar el lock (ver TTAS-Lock)

TTAS-Lock (Test and Test-And-Set Lock) (local spinning)

> El protocolo de Test-And-Set es complejo, y requiere bloquear la memoria para escritura. En vez de hacer spin (busy waiting) sobre la instrucción TAS, se hace sobre la instrucción get(), de forma tal que sólamente cuando el get() devuelva falso, se intentará hacer el TAS.

```
void create() {
    mutex.set(false);
}

void lock () {
    do {
        while (mutex.get()) {
            // busy waiting
        }
    } while (!mutex.testAndSet());
}

void unlock() {
    mutex.set(false);
}
```

Tiene más eficiencia/escalabilidad que TAS-Lock

- El while hace get() en lugar de testAndSet()
- Caché hit mientras el get es true
- Caché miss cuando hay un unlock

Otros objetos atómicos

> Read-Modify-Write Atómicos

```
/**
 * Obtiene un valor y lo incrementa en 1
 * (devuelve el valor original)
 */
atomic int getAndInc () {
    int tmp = reg;
    reg ++;
    return tmp;
}

/**
 * Obtiene un valor y le suma un entero
 * (devuelve el valor original)
 */
atomic int getAndAdd (int v) {
    int tmp = reg;
    reg = reg + v;
    return tmp;
```

```
}

/**
 * Compara contra el primer parámetro, si es igual
 * lo cambia por el segundo parámetro.
 * (devuelve el valor original)
 */
atomic T compareAndSwap (T u, T v) { // CAS
    T tmp = reg;
    if (u == tmp) reg = v;
    return tmp;
}

atomic enqueue(T item) {
    mutex.lock();
    queue.push(item);
    mutex.unlock();
}

atomic bool dequeue(T *pitem) {
    bool success;
    mutex.lock();
    if (queue.empty) {
        pitem = null;
        success = false;
    } else {
        pitem = queue.pop();
        success = true;
    }
    mutex.unlock();
    return success;
}
```

Semáforo

TODO: Revisar esta sección ¿Son atómicos? (rta en sistemas distribuidos)

- Una variable entera: capacidad
- Una cola de procesos en espera.
- wait() (P() o down()): Esperar hasta que se pueda entrar.
- signal() (V() o up()): Salir y dejar entrar a alguno.

```
void wait () {
// adquirir lock del kernel

    while (!capacidad) { // ocupado ( espera no acotada !!)
        fila.enqueue(self); // encolarse
        towaiting(self); // liberar lock y dormir
        // SIGNALED (recupera lock del kernel)
    }
    capacidad--;
```

```
// liberar el lock del kernel
}

void signal () {
// adquirir el lock del kernel

    capacidad++; // liberar semáforo
    if (q.dequeue(&p)) {
        toready(p); // despertarlo
    }

// liberar el lock del kernel
}
```

Mutex

> MUTual EXclusion

Se puede ver como un semáforo que sólamente toma los valores 0 o 1.

Modelo Coffmann (para detectar Deadlock)

> Serie de condiciones necesarias para la existencia de un deadlock

- Exclusión Mutua: existe un recurso que no puede ser asignado a más de un proceso
- Hold and Wait: los procesos pueden retener un recurso y solicitar otro
- No preemption: no hay un mecanismo compulsivo para quitarle los recursos a un proceso
- Espera circular: Tiene que haber un ciclo de N>=2 procesos, tal que P_i espera un recurso de P_i+1

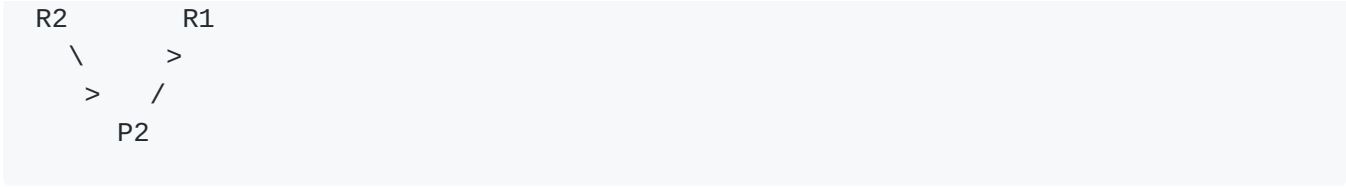
Modelo de Grafos Bipartitos (para detectar Deadlock)

> Hay deadlock cuando hay un ciclo

- Procesos: nodos P
- Recursos: nodos R
- Aristas:
  - **P->R** si P solicita R
  - **R->P** si P adquirió R

```
Ejemplo de deadlock

     P1
   /   <
  <     \
```



Problemas de sincronización

- Race condition (condición de carrera) El resultado no corresponde a ninguna secuencialización.
- Deadlock (bloqueo para siempre): Uno o más procesos quedan bloqueados para siempre.
- Starvation (innanición): Un proceso espera un tiempo no acotado para adquirir un lock porque otro(s) proceso(s) le gana(n) de mano.
- Convoying (demora en cascada): Un proceso que detiene un lock es sacado de running antes de liberar el lock.

Prevención

TODO: Revisar esta sección

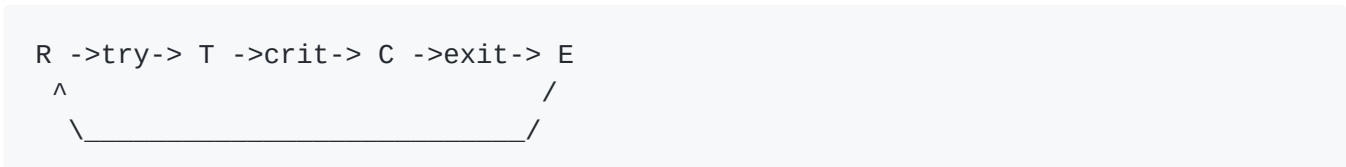
- Patrones de diseño
- Reglas de programación
- Prioridades
- Protocolos (e.g., Priority Inheritance)

Detección

TODO: Revisar esta sección

- Análisis de programas
  - Análisis estático
  - Análisis dinámico
- En tiempo de ejecución
  - Preventivo (antes que ocurra)
  - Recuperación (deadlock recovery)

Modelo teórico



- Estado: σ : [0 ... N−1] → {R,T, C, E}
- Transición: σ → σ', l ∈ {rem, try, crit, exit}
- Ejecución: τ = τ_0 → τ_1 ...
- Garantizar PROP: Toda ejecución satisface PROP
- Notación: #S = cantidad de elementos del conjunto S

Propiedades

Exclusión Mutua (EXCL)

> Para toda ejecución T y estado T_k, no puede haber más de un proceso i tal que T_k(i) = C.

- #CRIT ≤ 1

Progreso (PROG)

> Para toda ejecución τ y estado τ_k, si en τ_k hay un proceso i en T y ningún i' en C, entonces ∃j > k tal que en el estado τj algún proceso i' está en C.

- (#TRY ≥ 1 ∧ #CRIT = 0 =⇒ ♦#CRIT > 0)

Progreso global absoluto (WAIT-FREE)

> Para toda ejecución τ, estado τ_k y proceso i, si τ_k (i) = T entonces ∃j > k, tal que τ_j(i) = C

- ∀i. IN(i)
- IN(i): TRY (i) ⇒ ♦ CRIT(i)

Progreso global dependiente (G-PROG)

(deadlock-, lockout-, o starvation-free)

> Para toda ejecución τ, si para todo estado τ_k y proceso i tal que τ_k(i) = C, ∃j > k, tal que τ_j(i) = R entonces para todo estado τ_k' y todo proceso i', si τ_k'(i') = T entonces ∃j' > k' tal que τ_j'(i) = C.

- ∀i. OUT(i) ⇒ ∀i. IN(i)
- OUT(i): CRIT(i) ⇒ ♦ REM(i)

Justicia (FAIR) (fairness)

> Para toda ejecución τ y todo proceso i, si i puede hacer una transición L_i en una cantidad infinita de estados de τ entonces existe un k tal que τ_k →(L_i)→ τ_{k+1}.

Observaciones

- EXCL es una propiedad de safety: nada malo pueda pasar nunca.
- PROG, G-PROG y WAIT-FREE son propiedades de liveness: algo bueno debe pasar en el futuro.
- FAIR se asume: debe ser garantizada por el scheduler
- No hacer nada garantiza safety.
- Siempre hay que tener ambas propiedades.

Bibliografía

- Hoare, C. Monitors: an operating system structuring concept, Comm. ACM 17 (10): 549-557, 1974. http://goo.gl/eVaeeo
- Allen B Downey. The Little Book of Semaphores. http://goo.gl/ZB9zYl
- Edgar W. Dijkstra. Cooperating sequential processes. https://goo.gl/PqDzpm
- M. Herlihy, N. Shavit. The Art of Multiprocessor Programming. Morgan Kaufmann, 2008.
- N. Lynch. Distributed Algorithms. Morgan Kaufmann, 1996.
- Ch. Kloukinas, S. Yovine. A model-based approach for multiple QoS in scheduling: from models to implementation. Autom. Softw. Eng. 18(1): 5-38 (2011). https://goo.gl/5FuU6x
- M. C. Rinard. Analysis of Multithreaded Programs. SAS 2001: 1-19 http://goo.gl/pyfg0G
- L. Sha, R. Rajkumar, J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. IEEE Transactions on Computers, September 1990, pp. 1175-1185. http://goo.gl/0Qeujs
- Valgrind tool. http://valgrind.org/
- Java Pathfinder (JPF). http://babelfish.arc.nasa.gov/trac/jpf

Sincronización

- El problema de sincronización de procesos surge a partir del punto en que se los quiere ejecutar de forma concurrente.
  - Los procesos pueden ser interrumpidos, dejando ejecuciones parciales.
  - El acceso concurrente a los datos puede ocasionar inconsistencias.

Contención

- El problema es cuando los procesos comparten recursos, y acceden a los mismos de forma concurrente.

Race Condition / Condición de carrera

- El resultado de una ejecución concurrente varía según el orden en que se ejecutan las operaciones concurrentes.
- Dicho de otro modo, es cuando una ejecución concurrente se corresponde a más de una serialización, y estas producen distintos resultados.

Ejemplo de Race Condition:

```
Proceso 1:
    (1)     temp1 <- lee(A)
    (2)     temp1 <- temp1 + 1
    (3)     escribe(A) <- temp1

Proceso 2:
    (4)     temp2 <- lee(A)
    (5)     temp2 <- temp2 + 1
    (6)     escribe(A) <- temp2
```

Se establece el siguiente orden relativo entre las operaciones (dependencias): *(1) -> (2) -> (3): 2 se ejecuta luego de 1, y 3 se ejecuta luego de 2* (4) -> (5) -> (6): 5 se ejecuta luego de 4, y 6 se ejecuta luego de 5

```
Serialización 1:
    (0)     (Pre: A = 0)

    (1)     temp1 <- lee(A)         # temp1 = 0
    (2)     temp1 <- temp1 + 1      # temp1 = 1
    (3)     escribe(A) <- temp1     # A     = 1
    (4)     temp2 <- lee(A)         # temp2 = 1
    (5)     temp2 <- temp2 + 1      # temp2 = 2
    (6)     escribe(A) <- temp2     # A     = 2

    (7)     (Post: A = 2)

Serialización 2:
    (0)     (Pre: A = 0)

    (1)     temp1 <- lee(A)         # temp1 = 0
    (4)     temp2 <- lee(A)         # temp2 = 0
    (5)     temp2 <- temp2 + 1      # temp2 = 1
    (2)     temp1 <- temp1 + 1      # temp1 = 1
    (6)     escribe(A) <- temp2     # A = 1
    (3)     escribe(A) <- temp1     # A = 1

    (7)     (Post: A = 1)
```

- Ambas serializaciones son correctas, puesto que respetan las dependencias mencionadas previamente.
- Sin embargo, ¡ambas producen distintos resultados!

Critical Section / Sección Crítica

- En el caso anterior, el acceso a A debería ser protegido.
  - Proceso 2 no debería poder utilizar A mientras Proceso 1 lo esté utilizando.
  - Proceso 1 no debería poder utilizar A mientras Proceso 2 lo esté utilizando.
- La solución a esto, es establecer una sección crítica.
- Una sección crítica es un bloque de código en donde el acceso a un recurso compartido está protegido, de forma tal que sólamente un proceso puede entrar en esta sección a la vez.
- Normalmente una sección crítica se divide en tres partes:
  - **Entry Section / Entrada**: Donde el proceso solicita permiso para acceder a un recurso. Mientras no se le otorgue este permiso, el proceso va a permanecer en esta sección.
  - **Critical Section**: Es el código que utiliza el recurso compartido.
  - **Exit Section / Salida**: Donde el proceso entrega el permiso de acceso a ese recurso. De este modo, cualquier otro proceso que esté esperando para entrar a la sección crítica, puede hacerlo.
- Se considera también una **Remainder Section** como aquella sección del proceso que se encuentra totalmente por fuera de la sección crítica.
- Es importante que la sección crítica sea lo más pequeña posible.

Resolución de Sección Crítica

Se requieren satisfacer tres condiciones:

1. Exclusión mútua: Si el proceso Pi está ejecutando en su sección crítica, entonces ningún otro proceso puede ejecutar en la sección crítica correspondiente.
2. Progreso: Si ningún proceso está en su sección crítica, y existe algún proceso que desea entrar, se le debe garantizar su entrada a uno de ellos.
3. Espera Acotada: Ningún proceso debería esperar indefinidamente para entrar a su sección crítica, es decir, una vez que se encuentra encolado, su tiempo de espera se debería considerar acotado (sin deadlock, sin starvation).

Algoritmo de Peterson

```
El algoritmo de Peterson resuelve exclusión mútua para dos procesos.
```

```
do {
    ////////////////////////////////////
    // Entry Section
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);
    ////////////////////////////////////

    ////////////////////////////////////
    // Critical Section
    // ...
    ////////////////////////////////////

    ////////////////////////////////////
    // Exit Section
    flag[i] = false;
    ////////////////////////////////////

    ////////////////////////////////////
    // Remainder Section
    // ...
    ////////////////////////////////////
} while (true);
```

- flag[i] indica si el proceso "i" está interesado en entrar a la sección crítica.
- turn indica de quién es el turno de entrar en la sección crítica.

Peterson - idea

- Si un sólo proceso está interesado en entrar en la sección crítica, entonces entra.
  - Si mientras un proceso está en su sección crítica el otro proceso quiere entrar en la sección crítica, va a marcar su flag de interés en true.
  - Como el flag de interés del otro proceso va a estar en true, va a evaluar de quién es el turno.
  - Como cada proceso le cede el turno al otro, va a tener que esperar a que el otro termine.
- Una vez que el proceso que estaba en la sección crítica termina, pone su flag de interés en falso.
  - Si el otro proceso quería entrar a la sección crítica, lo va a poder hacer ahora.
- Si ambos procesos quieren entrar a la sección crítica "a la vez", alguno de ellos va a ganar (el primero que setee la variable turn, termina ganando acceso a la sección crítica).

Peterson - correctitud

- Se preserva exclusión mútua porque los dos procesos no pueden estar a la vez en la sección crítica.

- Se preserva progreso, porque si ambos quieren entrar a la vez (o si uno solo quiere entrar), va a terminar entrando.
- Se preserva espera acotada, porque si un proceso quiere entrar a la sección crítica, a lo sumo va a tener que esperar un turno (si entra el otro), y apenas termine el otro va a poder entrar.
    - Esto sucede incluso aunque el otro pueda hacer todo el loop y volver a entrar a la zona de entrada de sección crítica, ya que cada proceso le cede el turno al otro.

Peterson - Desventajas

- Está limitado a dos procesos.
- Realiza busy waiting.

Primitivas de Hardware

- Muchos sistemas de hardware proveen soluciones para implementar una solución al problema de la sección crítica.
- Normalmente se basan en locks.
- Sistemas mononúcleo, pueden simplemente deshabilitar interrupciones.
    - El código va a correr sin desalojo (preemption), ¡esto es un peligro!
    - No es eficiente, y no es sencillo de hacer en multinúcleo.
- Sistemas modernos, proveen instrucciones de hardware especiales
    - Instrucciones atómicas: **no pueden ser interrumpidas**, se realizan en un solo paso.
    - Test-And-Set: Obtiene un valor de memoria, y al mismo tiempo lo setea en true.
    - Compare-And-Swap: Intercambia dos posiciones de memoria.

Solución genérica a sección crítica usando locks

```
do {
    ////////////////////////////////////
    // Entry Section
    acquire_lock()
    ////////////////////////////////////

    ////////////////////////////////////
    // Critical Section
    // ...
    ////////////////////////////////////

    ////////////////////////////////////
    // Exit Section
    release_lock()
    ////////////////////////////////////
```

```
    ////////////////////////////////////
    // Remainder Section
    // ...
    ////////////////////////////////////
} while (true);
```

Test-And-Set

```
atomic boolean test_and_set (boolean *target)
{
   boolean rv = *target;
   *target = true;
   return rv:
}
```

Test-And-Set: Shared Boolean variable lock

```
do {
    ////////////////////////////////////
    // Entry Section
    while (test_and_set(&lock)) {
        /* do nothing */
    }
    ////////////////////////////////////

    ////////////////////////////////////
    // Critical Section
    // ...
    ////////////////////////////////////

    ////////////////////////////////////
    // Exit Section
    lock = false;
    ////////////////////////////////////

    ////////////////////////////////////
    // Remainder Section
    // ...
    ////////////////////////////////////
} while (true);
```

Test-And-Set: Bounded-waiting Mutual Exclusion

```
do {
    ////////////////////////////////////
    // Entry Section
    waiting[i] = true;
    key = true;
```

```
    while (waiting[i] && key){
        key = test_and_set(&lock);
    }
    waiting[i] = false;
    ////////////////////////////////////

    ////////////////////////////////////
    // Critical Section
    // ...
    ////////////////////////////////////

    ////////////////////////////////////
    // Exit Section
    j = (i + 1) % n;
    while ((j != i) && !waiting[j]){
        j = (j + 1) % n;
    }
    if (j == i){
        lock = false;
    }
    else{
        waiting[j] = false;
    }
    ////////////////////////////////////

    ////////////////////////////////////
    // Remainder Section
    // ...
    ////////////////////////////////////
} while (true);
```

Compare-And-Swap

```
atomic int compare_and_swap(int *value, int expected, int new_value) {
    int temp = *value;

    if (*value == expected) {
        *value = new_value;
    }
    return temp;
}
```

Compare-And-Swap: Shared integer “lock”

```
do {
    ////////////////////////////////////
    // Entry Section
    while (compare_and_swap(&lock, 0, 1) != 0); {
        /* do nothing */
    }
```

```
    ////////////////////////////////////

    ////////////////////////////////////
    // Critical Section
    // ...
    ////////////////////////////////////

    ////////////////////////////////////
    // Exit Section
    lock = 0;
    ////////////////////////////////////

    ////////////////////////////////////
    // Remainder Section
    // ...
    ////////////////////////////////////
} while (true);
```

Get-And-Inc

```
atomic int get_and_inc (int *target)
{
    int res = target;
    target++;
    return res;
}
```

Get-And-Add

```
atomic int getAndAdd ( int *target, int v )
{
    int res = *target;
    *target = *target + v;
    return res;
}
```

Spinlock / Test-And-Set lock / TASLock

Usando primitivas del Sistemas Operativo

```
atomic acquire() {
    while (!available) {
        /* busy wait */
    }
    available = false;
}
```

```
atomic release() {
    available = true;
}

do {
    acquire()
    /* Critical Section */
    release()
    /* Remainder Section */
} while (true);
```

Usando TAS

```
atomic bool testAndSet (reg) {
    bool tmp = reg;
    reg = true;
    return tmp;
}

available = false;

tas_acquire() {
    while (testAndSet(available)) {
        /* busy wait */
    }
}

tas_release() {
    available = false
}

do {
    tas_acquire()
    /* Critical Section */
    tas_release()
    /* Remainder Section */
} while (true);
```

Local Spinning / TTASLock

```
Variante del TASLock, que primero verifica la variable
```

```
atomic bool get (reg) {
    return reg;
}

atomic void set( reg, b ) {
```

```
    reg = b;
}

atomic bool testAndSet (reg) {
    bool tmp = reg;
    reg = true;
    return tmp;
}

mutex mtx;

void create () {
    mtx.set( false );
}

void lock () {
    while ( true ) {
        while (get(mtx)) {
            /* busy wait */
        }
        if (!testAndSet(mtx)) {
            return;
        }
    }
}

void unlock () {
    mtx.set( false );
}
```

Cola Atómica

```
private mutex mtx;
private queue q;

atomic enqueue (T item) {
    mtx.lock();
    q.push(item);
    mtx.unlock();
}

atomic bool dequeue (T * pitem) {
    bool success;
    mtx.lock ();
    if (q.empty()) {
        pitem = null;
        success = false;
    }
    else {
        pitem = q.pop ();
        success = true;
    }
```

```
    mtx.unlock();
    return success;
}
```

Mutex Recursivo

```
int calls;
atomic<int> owner;
private int self = 0x1234; // PID, o lo que sea

void create () {
    owner.set(-1);
    calls = 0;
}

void lock() {
    if (owner.get() != self) {
        while (owner.compareAndSwap(-1, self) != self) {
            /* busy wait */
        }
    }
    // owner == self
    calls ++;
}

void unlock () {
    if (--calls == 0) {
        owner.set(-1);
    }
}
```

Deadlock

```
Cuando por un conflicto de dependencias, la ejecución queda parada
Ejemplo:
    - el proceso A tiene el recurso 2, y espera el recurso 1
    - el proceso B tiene el recurso 1, y espera el recurso 2
    - ninguno libera sus recursos.
```

Modelo Coffmann (para detectar Deadlock)

```
Serie de condiciones necesarias
para la existencia de un deadlock
```

- Exclusión Mutua: existe un recurso que no puede ser asignado a más de un proceso

- Hold and Wait: los procesos pueden retener un recurso y solicitar otro
- No preemption: no hay un mecanismo compulsivo para quitarle los recursos a un proceso
- Espera circular: Tiene que haber un ciclo de N>=2 procesos, tal que P_i espera un recurso de P_i+1

Modelo de Grafos Bipartitos (para detectar Deadlock)

```
Hay deadlock cuando hay un ciclo
```

- Procesos: nodos P
- Recursos: nodos R
- Aristas:
  - **P->R** si P solicita R
  - **R->P** si P adquirió R

```
Ejemplo de deadlock

     P1
   /   <
  <     \
R2       R1
  \     >
   >   /
     P2
```

TODO: Completar

Registros Atómicos

Exclusión Mútua

- Teorema de Burns & Lynch: El mínimo número de registros atómicos necesarios para garantizar exclusión mútua sin restricciones de tiempo es N.
- Algoritmo de Fischer: Se asumen restricciones de tiempo. Suponiendo FAIRNESS, garantiza LOCK-FREEDOM si $\Delta > \delta$.

```c
void proceso(int i){
    // TRY
    while(turn!=i){
        waitfor(turn==0);
        turn = i;           // tarda a lo sumo T
        pause(W);       // espera un tiempo W > T
    }
```

```
    // CRIT
    …

    // EXIT
    turn = 0
}
```

- Algoritmo de Dijkstra: Garantiza EXCL. Suponiendo FAIRNESS, garantiza LOCK-FREEDOM, pero no WAIT-FREEDOM.

```
/*
Registros
        * flag[i]: atomic single-writer / multi-reader
        * turn: atomic multi-writer / multi-reader
*/

void proceso(int i){
    // TRY
    flag[i] = 1;
    while(turn != i) {
        if(flag[turn]==0){
            turn = i ;
        }
    }
    flag[i] = 2;

    foreach(j!=i) {
        if (flag [j] == 2) goto TRY;
    }

    /* CRIT */
    ...

    /* EXIT */
    flag[i] = 0;
```

}

- Panadería de Lamport:

```
/*
Registros:
    choosing[i], number[i]: atomic single-writer / multi-reader
*/

/* TRY */
choosing[i] = 1;
number[i] = 1 + max{j!=i}(number [j]);
choosing[i] = 0;
foreach j != i {
```

```
    waitfor choosing [j]==0;
    waitfor number [j]==0 || (number[i], i) < (number[j] , j);
}

/* CRIT */
...

/* EXIT */
number [i] = 0;
```

Consenso

- Teorema Herlihy & Lynch: No se puede garantizar consenso para un n arbitrario con registros RW atómicos.

Administración de Memoria

Es uno de los roles del Sistema Operativo, y por lo general es un subsistema propio.

La memoria se comparte: - IPC - Multiprogramación

Roles de un subsistema de memoria

- Manejar el espacio libre (y ocupado)
- Asignar (y liberar) memoria
- Swapping (memoria principal <-> disco)

Objetivos

- Si hay un solo proceso en memoria, no es necesario compartir nada.
- Si hay varios procesos corriendo concurrentemente:
  - Se puede hacer swapping siempre (cada proceso que no esté corriendo se pasa a disco). Esto es lento, e innecesario.
  - Mejor opción: se pueden dejar todos los procesos que se puedan en memoria, mientras entren en ella.
  - Si no hay espacio en la memoria, se hace swapping para desalojar los procesos que no estén corriendo. ¡Cuando se necesiten volver a alojar, el espacio de memoria que les toque seguramente no sea el mismo!

Problemas

- Manejo del espacio libre (evitando la fragmentación).

- Protección (memoria privada de los procesos).
- Reubicación (cambio de contexto, swapping).

Manejo del espacio libre / Fragmentación

Fragmentación

Fragmentación Externa

Cuando los bloques de memoria están dispersos, de forma tal que no es posible asignarle una determinada cantidad de memoria **contigua** a un proceso.

Ejemplo, se cuenta con el siguiente mapa de memoria:

```
| proceso A | libre | libre | proceso B | libre | libre | proceso C | libre |
```

Un proceso D necesita 3 bloques de memoria. El sistema cuenta con 5 bloques libres, por lo que debería poder asignarle la memoria al proceso. Pero debido a que la misma está fragmentada, no puede ser asignada (la máxima cantidad de bloques contiguos de la que dispone el sistema es 2).

Una solución sería compactar/reubicar los bloques:

```
| proceso A | libre | libre | proceso B | libre | libre | proceso C | libre |
pasaría a ser
| proceso A | proceso B | proceso C | libre | libre | libre | libre | libre |
```

Pero esto es costoso en tiempo, y a veces no es posible (ejemplo: sistemas RT).

Fragmentación Interna

Cuando se divide la memoria en bloques de tamaño fijo, un proceso necesita una cantidad X de memoria, y se le asigna una cantidad de memoria Y>X, de forma tal que el espacio (Y-X) se desperdicia, ya que el proceso no lo utiliza, y no puede ser asignado a otros procesos.

Organización de la Memoria

Mapa de Memoria de una Tarea

```
+ HIGHER ADDRESSES

------- <- bottom
 STACK      <- Variable Size (grows down --)
------- <- top
```

```
    .    \
    .     \
    .      | <- Free Space
    .     /
    .    /
-------
  HEAP      <- Variable Size (grows up ++)
-------
  BSS       <- Fixed Size
-------
  DATA      <- Fixed Size
-------
  TEXT      <- Fixed Size
-------

- LOWER ADDRESSES
```

Bitmap

Se divide la memoria en bloques de tamaño fijo. El bitmap es un arreglo de booleanos. Cada posición del bitmap representa un bloque, y se guarda 0 si está libre, 1 si está ocupado.

- Asignar y liberar bloques es sencillo O(1)
- Encontrar bloques consecutivos es O(N)
- Hay un tradeoff entre granularidad y el tamaño del bitmap:
  - bloques de menor tamaño -> menos fragmentación interna -> mayor tamaño de bitmap
  - menor tamaño de bitmap -> bloques de mayor tamaño -> más fragmentación interna
- No es muy usado

Lista Enlazada

Cada nodo representa a la memoria de un proceso, o a un bloque libre (en cuyo caso figura el tamaño del bloque)

- Liberar y asignar es O(1) (manejar punteros)
- Se puede combinar con Bitmap
- ¿Dónde asignar los bloques?
  - First Fit: usa el primer bloque libre.
    - Es rápido.
    - Genera fragmentación.
  - Best Fit: busca el bloque más adecuado (el más pequeño posible)
    - Es más lento.
    - ¡...no es mejor que First Fit!
  - Quick Fit: variación de Best Fit.
    - Se mantiene una lista de los bloques libres de los tamaños más solicitados.

        - No es tan lento como Best Fit.
    - Buddy System
        - Usa splitting* de bloques.

Todos los esquemas anteriores fallan, produciendo fragmentación (externa o interna). En la práctica se usan esquemas más sofisticados, y no son triviales (¡están fuera del alcance de la materia!).

Memoria Virtual

TODO: 07-teorica-administracion-de-memoria.pdf, página 16/44

Entrada / Salida

```
Transferencia de datos desde y hacia dispositivos externos.
```

Tipos de dispositivos

- Almacenamiento (discos, cdrom, pendrive, ...)
- Comunicaciones (placa de red, modem, ...)
- Interface de usuario (teclado, mouse, monitor, ...)
- Otros

Dispositivos de Almacenamiento

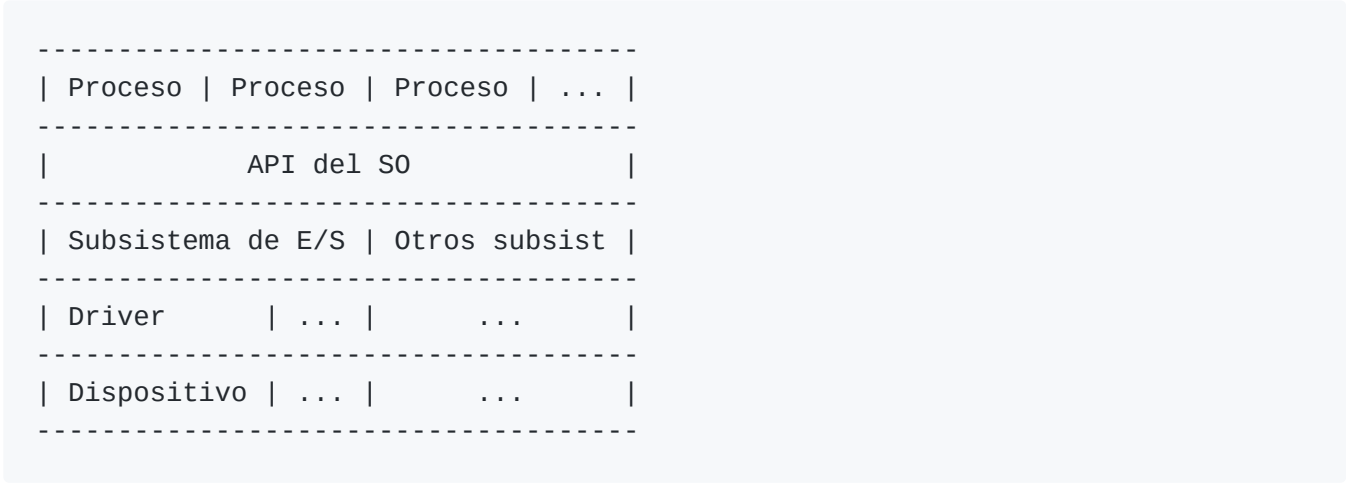
- Discos rígidos
- Unidades de cinta
- Discos removibles
- Unidades de red

Esquema de un dispositivo (en capas)

Desde el Hardware

- Dispositivo físico: es el dispositivo en sí.
- Controlador del dispositivo: es el hardware que interactúa con el SO mediante un bus o registro.
- Bus: es el hardware comunica el controlador de dispositivo con la CPU, muchas veces tiene un controlador.

Desde el Software



Drivers

- Son componentes de software muy específicos.
- Saben comunicarse a bajo nivel con el hardware particular.
- Por lo general, cada hardware requiere un Driver específico.
- Son muy importantes:
  - Forman parte del kernel, o se incorporan como módulos
  - Corren con máximo privilegio
  - El rendimiento de E/S depende de ellos
  - Un driver mal programado puede repercutir en el rendimiento (!)
  - Un driver mal programado puede colgar el sistema operativo (!)

Formas de interactuar con el Hardware

Polling

- El driver verifica cada un tiempo X si el dispositivo se comunicó
- Ventajas: sencillo, fácil de implementar, cambios de contexto controlados (pensar en un SO RT)
- Desventajas: CPU-intensivo, puede representar un desperdicio de recursos

Interrupciones (o push)

- El dispositivo avisa al SO que se quiere comunicar (mediante una interrupción)
- Ventajas: eficiencia en la utilización de CPU; sólamente se avisa cuando sucede un evento real
- Desventajas: más difícil de implementar, los cambios de contexto son impredecibles

DMA

- El dispositivo accede (lee/escribe) directamente a memoria.
- La CPU no interviene en el acceso a esa memoria.
- Cuando el dispositivo termina, avisa a la CPU mediante una interrupción.
- Ventaja: se pueden transferir grandes volúmenes de datos, sin necesidad de que la CPU intervenga.
- Desventaja: necesita hardware adicional (controlador de DMA)

Subsistema de E/S

- Brinda a los procesos API sencilla:
  - open() / close()
  - read() / write()
  - seek()
- A pesar de lo anterior, hay cosas que no se deben ocultar.
  - Ejemplo: acceso **exclusivo** al dispositivo.
- El SO tiene que hacer todo esto de manera *correcta y eficiente*
- La responsabilidad se comparte entre el Manejador de E/S y el Driver (!)

Char Device

```
La información se transmite byte a byte
```

- No soportan acceso aleatorio.
- No utilizan caché.
- Ejemplo: Mouse, teclado, tty, puerto serie, ...

Block Device

```
La información se transmite en bloques
```

- Permiten acceso aleatorio.
- Pueden utilizar un caché.
- Ejemplo: disco rígido, cdrom, ...

Subsistema de E/S Linux:

```
Acceso mediante /dev
```

Características:

- lectura, escritura, o lecto-escritura
- secuenciales o aleatorios
- compartidos o dedicados
- char o block
- sincrónicos o asincrónicos
- tienen distinta velocidad de respuesta

Una de las funciones del SO es brindar mediante su API un acceso *consistente*, ocultando en la medida de lo posible las particularidades de cada uno de los dispositivos.

API del subsistema de E/S de Linux:

- Todo es un archivo (!)
- Se proveen funciones de alto nivel para el acceso:
  - fopen, fclose: abrir y cerrar un descriptor
  - fread, fwrite: leer y escribir en modo **bloque**
  - fgetc, fputc: leer y escribir en modo **char**
  - fgets, fputs: leer y escribir en modo **char stream**
  - fscanf, fprintf: leer y escribir en modo char con formato

Planificación de E/S:

- Una de las claves para tener un buen rendimiento en un disco es manejarlo correctamente. Ejemplo, hay un cabezal que se mueve, y eso lleva tiempo. Queremos minimizar esos movimientos.
- Los pedidos a E/S llegan constantemente, incluso antes de terminar con los que se estén ejecutando.
- La planificación de E/S se trata de cómo manejar la **cola de pedidos** para lograr el mejor rendimiento posible.
  - ancho de banda: máxima cantidad de bytes que se pueden transferir por unidad de tiempo
  - latencia rotacional: tiempo necesario para que el disco rote hasta el sector deseado
  - seek time: tiempo para que la cabeza se ubique sobre el cilindro deseado (!)

Políticas de scheduling de E/S a disco:

- FIFO / FCFS: First Come, First Served
- SSTF: Shortest Seek Time First (!)
  - El próximo pedido es el más cercano.

    - ¡Puede producir inanición! (!)
    - Además, no es óptimo.
- Scan / Ascensor: Ir en un sentido, y luego en el otro.
    - Puede producir casos patológios (ej: siempre llegan pedidos en la posición inmediata anterior, maximizando el tiempo de espera)
    - El tiempo de espera no es uniforme.
- En la práctica, ninguno de estos se utiliza de forma pura.
- Se usan híbridos.
- Además, hay lecturas y escrituras que son prioritarias sobre otras (bajar páginas de caché, swapping de procesos)

SSD: Solid State Drive

```
Mejores prestaciones, menor precio.
```

- Ventajas:
    - No tienen componentes mecánicos.
    - Más livianos, resistentes y silenciosos.
    - Menor consumo eléctrico.
    - Mejor performance que los HDD.
- Problemas:
    - La escritura es más compleja que en un HDD.
    - Durabilidad
    - Write Amplification

Spooling

```
Simultaneous Peripheral Operation On-Line
Manejar acceso dedicado en sistemas multiprogramado
```

- Caso típico: la impresora.
- El acceso es exclusivo.
- No queremos que un proceso se bloquee al intentar un acceso sobre un dispositivo que está siendo utilizado.
- Se pone el trabajo en una cola, y se designa un proceso específico que va desencolándolos a medida que se libera el dispositivo.
- Para el Kernel no hay ninguna diferencia, no puede distinguir si se hace spooling o no.
- Para el usuario sí hay una diferencia (!)

Otros usos: Locking

- POSIX garantiza que open(..., O CREAT | O EXCL) es atómico, y falla si el archivo ya existe
- Esto brinda un mecanismo de exclusión mutua
- Se puede usar para implementar locks a nivel E/S

Protección de la información

- Hay un tradeoff entre el valor de la información y el costo de protegerla (!)
- Una estrategia muy utilizada es asumir que no se va a romper. ¡Es muy mala!

Copias de Seguridad (backup)

```
Resguardar los datos importantes en otro lado
```

- Se suelen hacer en cintas.
- También se puede copiar los datos a otro disco (local, por red, etc).
- Como sea, toma tiempo, y por eso se suele hacer durante la noche.
- Copiar todos los datos es costoso.
  - Una vez a la {semana/mes/etc} hacer un **backup total**
  - Realizar períodicamente un **backup incremental**: sólo los archivos modificados desde la última copia incremental (menos espacio / más procesamiento).
  - Otra opción es hacer períodicamente un **backup diferencial**: sólo los archivos modificados desde la última copia total (más espacio / menos procesamiento).
- Restaurar los datos también puede ser costoso.
  - Si sólamente hago copias totales, basta con tomar el backup de ese día.
  - Si hago copias diferenciales, tomo el último backup total, y el backup diferencial de ese día, y los junto.
  - Si hago copias incrementales, tengo que tomar el último backup total, y todos los backups incrementales que hayan habido hasta ese día, y juntarlos. ¡Esto es costoso!

Redundancia

```
Prevenir que el sistema quede fuera de línea
```

- Método muy común: RAID (Redundant Array of Inexpensive Discs)
- La idea (en general) es usar más de un disco, en donde se replica la información; si alguno se rompe, los otros sirven de resguardo.

Métodos:

- Raid 0 / Stripping
    - Por lo general un mismo archivo está distribuido en varios discos
    - La lectura y la escritura se pueden hacer en paralelo
    - Mejora el rendimiento
    - No aporta redundancia
- Raid 1 / Espejo / Mirror:
    - Los discos son copias idénticas el uno del otro.
    - Mejora el rendimiento de la lectura.
    - Las escrituras están supeditadas en el caso general al disco más lento. Además, en el mejor caso tardan lo mismo, en el peor caso tardan el doble.
    - Es muy caro de implementar, ya que para una misma cantidad de información requiere comprar el doble de discos.
- Raid 0+1:
    - Combina los dos anteriores.
    - Es un raid 0, al que se le hace Espejado.
    - Cada archivo está espejado, pero al leerlo leo un bloque de cada disco.
    - Esto significa que cada raid va a tener toda la información.
    - Es más rápido que mirroring.
- Raid 1+0 / Raid 10:
    - Combina los dos anteriores, pero al revés que Raid 0+1.
    - Es un raid 1, al que se le hace Stripping.
    - Cada bloque está repartido en un raid distinto.
    - Cada raid va a tener una parte de la información.
    - Es la configuración más rápida después de RAID 0.
- Raid 2 y 3:
    - Por cada bloque, se guarda información que permite determinar si hubo un daño o no.
    - Ejemplo: paridad.
    - Los errores se pueden corregir, recomputando el bloque dañado a partir de la información redundante.
    - Cada bloque lógico se reparte entre todos los discos.
    - RAID 2 requiere 3 discos de paridad por cada 4 discos de datos.

    - RAID 3 requiere 1 disco de paridad.
    - Todos los discos participan de todas las operaciones de E/S, por lo que es más lento que RAID 1.
    - Además, requiere un procesamiento adicional para computar la redundancia.
    - Se suele implementar por hardware, con una controladora dedicada.

- Raid 4:

    - Es como RAID 3, pero hace stripping a nivel de bloque (cada bloque en un solo disco).
    - Cada operación de bloque requiere un solo disco, más el disco de paridad, que participa de todas las operaciones.
    - Entonces, el disco de paridad sigue siendo un cuello de botella para el rendimiento.

- Raid 5:

    - Junto con raid 0, 1 y 0+1 es uno de los más usados en la práctica.
    - También discos usa discos redundantes, pero los distribuye en todos los discos.
    - No hay un sólo disco que tenga toda la redundancia.
    - Para cada bloque, algún disco tiene los datos, y algún otro tiene la paridad.
    - No hay cuello de botella para las escrituras si se mantiene la paridad distribuída.
    - ¡Soporta la pérdida de cualquiera de los discos!
    - Cuando se pierde un disco y se reemplaza, la reconstrucción afecta el rendimiento notablemente.
    - Se suele utilizar con un **HOT SPARE**: esto es un disco de reserva, que si bien no funciona normalmente con el RAID, sirve para reemplaza de forma rápida y automática cuando se daña uno de los discos.

- Raid 6:

    - Como RAID 5, pero con un 2do bloque de paridad.
    - El objetivo es soportar la rotura de hasta 2 discos.
    - Como RAID 5 se suele utilizar con hot spare, no hay una gran diferencia de "espacio desperdiciado".

Peligros de RAID

- No protege contra eliminación o modificación **accidental** de archivos
- Tampoco protege contra malware, o eliminación/modificación **intencional** de archivos (!)
- ¡Se combina con copias de seguridad! Una cosa no quita la otra.
- Si una aplicación corrompe datos, ninguno de los enfoques sirve.
- Si se corrompe la estructura de los archivos, o el filesystem, ninguno de los enfoques sirve.

- Los sistemas de archivos brindan su propia capa de protección.

Sistemas de Archivos

Archivo:

- Secuencia de bytes.
- Sin estructura interna (a priori).
- Se los identifica con un nombre.
- El nombre puede incluir una extensión.
  - archivo.txt: archivo de texto
  - archivo.tex: archivo de LaTeX
  - archivo.c: archivo de código fuente en C

Atributos de un Archivo:

- Nombre: ejemplo "archivo.txt"
- Identificador: identificador único de archivo, interno al sistema operativo
- Tipo: si pueden haber distintos tipos de archivo (ejemplo archivo o carpeta), es necesario identificarlos
- Ubicación: dónde está guardado el archivo (dispositivo) y en qué ruta.
- Tamaño: el tamaño del archivo
- Proteción: ACL, usuarios, permisos (lectura, escritura, ejecución), etc
- Fechas: creación, modificación, acceso, etc
- Atributos extendidos: checksum, encoding (ej: TF8), etc
- Toda esta información se almacena en el **Directory Structure** del FileSystem.

Operaciones de un Archivo:

```
El archivo se puede ver como un tipo abstracto de datos.
En ese sentido, es definido a través de sus operaciones.
```

Ejemplos: *create: crear el archivo* read: leer de un archivo *write: escribir en un archivo* seek: moverse a través de un archivo *delete: eliminar un archivo* truncate: truncar un archivo (eliminar a partir de una posición) *append: escribir en un archivo a partir del final* rename: renombrar un archivo *copy: copiar un archivo* lock: bloquear un archivo (para que otros no lo usen)

Filesystems:

```
El filesystem es un módulo del kernel.
```

- Algunos SO soportan sólamente un filesystem (DOS sólo soporta FAT)
- Otros soportan más de uno (Windows soporta FAT, FAT32 y NTFS)
- Otros soportan algunos, pero pueden soportar más mediante **módulos dinámicos** (ej Unix modernos)

Filesystem populares:

- Locales:
  - FAT / FAT32: popular en DOS y Windows viejos
  - NTFS: popular en Windows basados en NT
  - UFS / Unix file system / FFS / BSD Fast File System: popular en BSD
  - ext2 / ext3 / ext4: los más populares en Linux
  - XFS: 64 bits, alta performance
  - ZFS: es al mismo tiempo un filesystem y un LVM; incluye conceptos como integridad, redundancia (RAID-Z), compresión, snapshots, ACLs, etc.
  - RaiserFS: journaled, el creador mató a la esposa xdxd
  - ISO-9660: el filesystem de los CDROM
- De red:
  - NFS: Network File System, es un protocolo para sistemas de archivo de red
  - SMBFS / CIFS / SAMBA: Server Message Block / Common Internet File System: el protocolo para el filesystem de red de Windows, que también tiene una implementación libre
  - DFS: Distributed File System, protocolo que permite, en líneas generales, agrupar varios filesystem de red (ejemplo varios SMB) en uno solo.
  - AFS / CodaFS / etc: Andrew File System, just another distributed file system...

Responsabilidades de un Filesystem:

- **Organización lógica de los archivos (visible desde fuera).**
  - Organización interna: cómo se estructura la información dentro del archivo.
    - Ejemplo, en Windows y Unix por lo general se usa secuencia de a bytes. Esto significa que la responsabilidad de organizar el archivo internamente es del usuario.
  - Organización externa: cómo se ordenan y distribuyen los distintos archivos.

        - Hoy en día la mayoría de los Filesystem adhieren al concepto de directorios, conformando una organización jerárquica, con forma de arbol.
        - Casi todos soportan, además, una noción de link: un alias para un mismo archivo (es decir, el mismo archivo físico aparece como dos archivos lógicos distintos).
        - Con esta última noción, la estructura deja de ser un árbol, y se transforma en un grafo dirigido, introduciendo la posibilidad de que existan ciclos.

- **Nombre y ruta de los archivos.**
    - Caracteres de separación de directorios, ejemplo "/" (Linux) o "\" (Windows).
    - Extensión de los archivos.
    - Restricciones de longitus, y caracteres permitidos (ejemplo acentos).
    - Distinción entre mayúsculas y minúsculas (case sensitive o insensitive).
    - Prefijado o no por el equipo en el que se encuentran.
    - Ejemplos:
        - /usr/local/etc/apache.conf
        - C:\Program Files\Antivirus\Antivirus.exe
        - \SERVIDOR3\Parciales\parcial1.doc
        - servidor4:/ejercicios/practica3.pdf

- **Punto de montaje.**
    - A partir de qué nombre en la ruta se considera que un dispositivo está montado.
    - En Unix, se realiza mediante el comando mount.
    - Ejemplo: `mount /dev/sda1 /media/disco1` va a montar la primera partición del disco `sda` en la ruta `/media/disco1`. O sea que si existe el archivo `/carpeta/pepito.txt` en este disco, se va a poder acceder mediante la ruta `/media/disco1/carpeta/pepito.txt`.

- **Representación del archivo (¡Importante!)**
    - Esto es la organización lógica interna (no visible desde fuera del filesystem)
    - ¿El archivo está todo junto, o se divide en partes?

- **Gestión del espacio libre.**

- **Metadatos.**

La forma de manejar muchas de estas responsabilidades determinan las características, las ventajas y desventajas, de cada filesystem, principalmente en cuanto a su rendimiento, usabilidad y confiabilidad.

Representación de archivos

- Un archivo es una lista de bloques + metadatos.
- La forma más sencilla de representarlo es poner todos los bloques juntos en el disco.
    - Las lecturas son muy rápidas...
    - ¡Genera fragmentación externa!
    - Si el archivo crece y no tiene espacio para hacerlo, debe ser relocalizado.
    - Las escrituras, en el caso general, terminan siendo lentísimas.
    - Nadie usa este esquema.
- Otra forma de representar es mediante una lista enlazada de bloques.
    - Las lecturas consecutivas son rápidas.
    - Las lecturas aleatorias son muy lentas.
    - Se desperdicia un espacio constante en cada bloque para indicar cuál es el siguiente.
- **Solución de DOS, FAT:**
    - Una forma de mejorar esto es mediante una tabla global que me indique, para cada bloque, dónde está el bloque siguiente.
    - No desperdicio espacio interno del bloque.
    - Al tener la tabla en memoria, puedo leer los bloques fuera de orden.
    - Las lecturas aleatorias no son tan lentas.
    - Tengo que tener toda la tabla en memoria, no escala.
    - Hay una única tabla, genera contención (cuello de botella / todo depende de un mismo recurso).
    - Es poco robusto, toda la tabla está en memoria.
    - Además, no maneja permisos/seguridad.
- **Solución de Unix, Inodos:**
    - Cada archivo tiene un inodo.
    - El inodo tiene metadatos / atributos (tamaños, permisos, etc).
    - También tiene las direcciones de algunos bloques de acceso directo, permitiendo acceder rápidamente a archivos pequeños (ejemplo, con bloques de 8KB, se puede acceder hasta 96 KB).
    - Tiene una entrada llamada Single Indirect Block, que apunta a un bloque que contiene punteros a bloques de datos. Siguiendo el ejemplo anterior, esto permite manejar archivos de hasta 16 MB.

    - Tiene otra entrada llamada Double Indirect Block, que apunta a un bloque que contiene punteros a Single Indirect Blocks. Permite manejar archivos de hasta 32 GB.
    - Tiene otra entrada llamada Triple Indirect Block, que apunta a un bloque que contiene punteros a Double Indirect Blocks. Permite manejar archivos de hasta 70 TB.
    - Permite tener en memoria sólamente los inodos correspondientes a archivos abiertos.
    - Genera menos contención.
    - Consistencia/robustez, sólo están en memoria los inodos de los archivos abiertos.
    - **Inodos: Directorios**
        - Un inodo es una entrada al **ROOT DIRECTORY**.
        - Por cada archivo en ese directorio, hay una entrada.
            - En ese bloque se guarda una lista de (inodos, nombre de archivo).

TODO: PROFUNDIZAR ESTO CON BIBLIOGRAFIA

Inodos: Links

TODO: PROFUNDIZAR ESTO CON BIBLIOGRAFIA

Atributos / Metadatos de un filesystem

```
Por lo general son los atributos de los archivos
y además atributos particulares que pueda tener el FileSystem
```

- Permisos (default y ACL)
- Tamaño
- Propietario
- Fechas de creación, modificación acceso
- Tipo de archivo (regular, dispositivo virtual, pipe, socket, etc)
- Flags
- Conteo de referencias
- CRC o similar

Manejo del espacio libre

- Una posibilidad es utilizar un mapa de bits de bloques, donde 1 significa libre.
    - Calcular el espacio libre cuesta O(N), donde N es la cantidad de bloques.

    - Pero al ser mapa de bits, se puede comparar un registro entero (32 o 64 bits), y si es cero, se lo saltea
    - Esto hace que se puedan ahorrar comparaciones, pero no deja de ser una optimización.
- También es posible tener una lista enlazada de bloques libres.
- Por lo general se clusteriza.
    - La idea de esto es tener una lista de nodos que adentro tengan varios punteros a bloques.
    - TODO: PROFUNDIZAR ESTO CON BIBLIOGRAFIA

Caché

```
Se copia cierta información que está en disco, a memoria principal.
```

- Mejora el rendimiento
- En la práctica, se utiliza junto con el mecanismo de paginación (cada página en sí es una caché).
- Puede grabar las páginas de manera ordenada, ayudando a la planificación de escritura de IO.
    - Las aplicaciones pueden requerir la *escritura sincrónica*, es decir, grabar los cambios de forma inmediata. Obviamente, esto es más lento.

Consistencia

```
Es importante asegurar la coherencia/consistencia en las operaciones del disco
ante un evento inesperado (ejemplo: corte de luz), el filesystem puede quedar
en un estado inconsistente (operaciones fuera de orden, inodos a medio escribir,
```

- Si se quiere desmontar un filesystem, primero se deben aplicar todas las operaciones pendientes (en caché).
    - Para ello, el Sistema Operativo provee `fsync`, que se encarga de grabar todas las páginas sucias de la caché.
    - Para saber si una página de caché está sucia, basta con ver el bit correspondiente `dirty`.
- En ext existe un bit de consistencia, que se marca cuando se monta el filesystem, y se desmarca cuando el mismo es desmontado correctamente (ejemplo, ante un apagado normal de la PC).
- Cuando existe algún evento que produce un desmontaje imprevisto del filesystem (ej un corte de luz), este bit queda marcado.

- Cada vez que se quiere montar el FS, se chequea previamente que el mismo esté en un estado coherente, mirando el bit en cuestión.
- En caso contrario, no se lo monta sin antes realizar un chequeo y restauración de consistencia en el mismo.
  - En Unix, esto se hace mediante el programa **fsck**.
  - Este programa recorre todo el disco, verifica consistencia (ej CRC) de cada inodo, y por cada bloque de datos cuenta cuántos inodos le apuntan, y también se fija si el mismo aparece referenciado como un bloque libre.
  - Hacer todo esto puede llegar a ser muy lento.
- Otra opción es hacer **Soft Updates**.
  - Permite iniciar el Sistema Operativo de forma mucho más rápida.
  - Mantiene un registro de operaciones/dependencias.
  - Este es flusheado períodicamente a disco.
  - Cuando se detecta un filesystem inconsistente, se busca la última secuencia de operaciones válidas en este registro. Esto es, aquellas operaciones que no tengan dependencias sin flushear.
  - Cualquier otra operación que haya sido flusheada, pero cuyas dependencias no se cumplan, es revertida.
  - Luego, ya se puede montar el filesystem e iniciar el sistema.
  - Resta chequear el listado de bloques libres, para verificar que no haya quedado ningún bloque marcado como ocupado, y que en realidad esté libre. Pero esto se puede hacer con el sistema andando.

Características Avanzadas

Cuotas de disco

```
Limitar cuánto espacio puede utilizar cada usuario
```

- Puede llegar a ser difícil de implementar.
- Soft Quota: Se avisa al usuario que está sobrepasando el límite, pero se le permite seguir violándolo.
- Hard Quota: Se le bloquea todo tipo de escritura al usuario.

Encriptación

- ¿Cómo y dónde guardar la clave?
  - Ejemplo: Clustered File System: ¿la clave se distribuye por los nodos?
- ¿Cómo desencriptar la clave? ¿Si pierdo la clave, pierdo todo el filesystem?
- Esto agrega un overhead de procesamiento.

    - **Más lento (!)**

Snapshots

- Son fotos del estado del disco en un determinado momento.
- Se realizan instantáneamente (o casi).
- El sistema operativo, luego, va a duplicar sólamente los archivos que sean modificados.
- Es muy útil para hacer backups.

Raid por Software

- Mayor control.
- Independencia del proveedor.
- Agrega un overhead de procesamiento
    - **Más lento (!)**
- Se agregan nuevas posibilidades y/o niveles de redundancia (ej: ZFS)

Compresión

- Los datos se guardan comprimidos.
- El driver del FS actúa como capa intermedia entre el dato físico comprimido, y el dato que se lee (descomprimido).
- Esto agrega un overhead de procesamiento.
    - **Más lento (!)**

Performance de un filesystem

El rendimiento de un FS se ve afectado por muchos factores.

- Tecnología de disco (hdd vs sdd, velocidad de lectura secuencial/aleatoria, velocidad de escritura, etc).
- Política de scheduling de IO.
- Tamaños de bloque.
- Cachés del SO (ej de IO).
- Cachés de las controladora (ej de SATA).
- Cachés de los propios discos.
- Sincronización / manejo de locks en IO.
- Journaling vs. softupdates.
    - TODO: PROFUNDIZAR ESTO CON BIBLIOGRAFIA
- Tradeoffs: hay cosas por las que vale la pena sacrificar performance
    - Seguridad

- Estabilidad
- Robustez
- Mantenibilidad

NFS: Network File System

- Permite acceder a un FS remoto, como si fuera local.
  - Utiliza RPC.
  - Esto se hace de forma transparente.
  - El NFS se monta en un mountpoint, como cualquier otro FS, y las aplicaciones no saben que es remoto.
- Para funcionar, requiere de una capa adicional del SO, llamada VFS (Virtual File System).
  - Esta capa tiene *vnodes* por cada archivo abierto.
  - Cada vnode se corresponde con un inode si el archivo es local.
  - Si el archivo es remoto, se almacenan metadatos adicionales.
  - De este modo, los pedidos que llegan al VFS son despachados al FS real, o al NFS.
  - Cada FS a su vez se encarga de los detalles implementativos del pedido (ej, NFS maneja el protocolo de red necesario para enviar el RPC)
- Es similar a otros sistemas distribuidos.
- NFS no es 100% distribuído.
  - Todos los datos de un mountpoint deben pertenecer al mismo "medio físico".
  - AFS o Coda son realmente distribuídos.

EXT2

TODO: PROFUNDIZAR ESTO CON BIBLIOGRAFIA

Sistemas Distribuídos

```
Conjunto de recursos interconectados
que interactúan entre sí.
```

Ejemplos:

- Varias máquinas conectadas en red.
- Un procesador con varias memorias.
- Varios procesadores con una memoria.
- Cualquier combinación de las anteriores.

- Etcétera.

Ventajas:

- Paralelismo: el trabajo se reparte, por lo que se hace más rápido.
- Replicación: la misma información o trabajo puede estar en varias partes, aumentando la confiabilidad.
- Descentralización: previene la contención.

Problemas:

- Sincronizar las operaciones.
- Mantener coherencia.
- Información parcial.
- No suelen compartir clock.
  - No se puede trabajar de forma sincrónica.

Sistemas con Memoria compartida

```
Se trabaja con algún mecanismo de Memoria Compartida, de forma tal que el Softwa
```

Por Hardware

- Uniform Memory Access (UMA)
- Non Uniform Memory Access (NUMA)
- Híbrida

Por Software

- Estructurada
  - Memoria Asociativa.
  - Distributed Arrays.
- No estructurada
  - Memoria Virtual global.
  - Memoria Virtual particionada.

Sistemas sin Memoria Compartida

```
Se depende de la cooperación del software para realizar las operaciones necesaria
```

Sincrónicos

- Telnet: es un protocolo para conectarse remotamente a un equipo, pero sirve como ejemplo de un programa delegando una tarea en una terminal remota. Entonces, los recursos invoulcrados (CPU, Memoria, etc) no son los locales, sino los de la máquina remota. En la máquina cliente, sólamente se requiere correr el propio protocolo de conexión, el cual es muy liviano.
- RPC (Remote Procedure Call): Es un mecanismo que permite a los programas llamar a funciones de manera remota (en un principio diseñado para C). Involucra bibliotecas que ocultan los detalles de la comunicación al programador, y permiten realizar el envío de datos de forma transparente. Es un mecanismo sincrónico.
    - Java Remote Method Invocation (RMI)
    - JavaScript Object Notation (JSON-RPC)
    - Simple Object Access Protocol (SOAP)
- Este tipo de arquitecturas se suelen llamar **cliente/servidor**.
    - El servidor proporciona servicios cuando el cliente se lo pide.
    - El servidor no tiene un rol activo en el procesamiento, simplemente espera que se le pidan cosas.
    - Para completar la tarea, entonces, el programa hace de cliente de los distintos servicios/servidores que va necesitando.

Asincrónicos

- RPC asincrónico
    - Promises
    - Futures
    - Windows Asynchronous RPC
- Pasaje de mensajes (send / receive)
    - Mailbox
    - Pipe
    - Message Passing Interface (MPI) para C/C++
    - Scala actors: send, receive/react

Pasaje de Mensajes

- Es el mecanismo más general.

- No asume que haya nada compartido, excepto el canal de comunicación.
  - Al haber un canal de comunicación, si el mismo serializa las operaciones, puede no ser necesario contar con Mutex.
- Desafíos
  - Codificación de los datos
  - Hay que dejar de procesar para atender los llamados de mensajes
  - Es lento
  - El canal puede perder mensajes
  - Podría haber un costo económico relacionado a la transmisión de cada mensaje (ej. supongamos que cada mensaje es un SMS).
- Existen bibliotecas que ayudan con estos problemas, la más popular es MPI.
- Curiosidad: **Conjetura de Brewer**, en un sistema distribuído no se puede tener a la vez consistencia, disponibilidad y tolerancia a fallas.

Locks en Sistemas Distribuídos

- No existe TestAndSet atómico (!)
- ¿Cómo se implementan los locks?

Centralizado

- Poner el control de los recursos (¡no los recursos en sí!) bajo un único nodo (coordinador).
  - Esto genera contención
- Dentro de ese nodo hay procesos que ofician de representantes (proxies) de los procesos remotos.
  - Es decir, este nodo central recibe los pedidos, e internamente los delega a los otros nodos, y viceversa.
- Cuando un proceso necesita un recurso, se lo pide al proxy, que se encarga de negociar con los otros proxies utilizando los mecanismos de sincronización habituales (al ser procesos locales).
- Problemas:
  - Todo depende de un nodo.
  - Hay un punto único de falla.
  - Cuello de botella en procesamiento y red.
  - Se requiere consultar al coordinador incluso para acceder a los propios recursos, o aquellos cercanos.
  - Cada interacción con el coordinador requiere de mensajes, lo cual agrega overhead.

Lamport

Idea general

- No importa cuándo suceden exactamente las cosas.
- Lo único importante es si algo ocurre antes o después de otra cosa.
- Se define un **orden parcial no reflexivo**.
  - Si dentro de un proceso A sucede antes que B, entonces A -> B
  - Si E es el envío de un mensaje y R es su recepción, E -> R. Incluso si E y R suceden en procesos distintos.
  - Si A -> B y B -> C, entonces A -> C.
  - Si no vale A -> B y tampoco vale B -> A, entonces A y B son concurrentes.

Implementación:

- Cada procesador tiene un reloj; tiene que ser un valor monótonamente creciente en función de las lecturas.
- En cada mensaje se incluye el valor del reloj.
- Como la recepción siempre es posterior al envío, al recibir un mensaje con una marca de tiempo T, con T mayor al reloj actual, se actualiza el reloj a T+1.
- Todo lo anterior da un orden parcial. Para generar un orden total, hay que desempatar.
  - Es necesario buscar una forma de ordenar arbitrariamente.
  - Por ejemplo, a través del PID.

Acuerdo Bizantino

```
El problema de nunca poder estar seguro de que el mensaje (o la confirmación, o l
```

Teoremas

- No existe ningún algoritmo para resolver consenso en un escenario de fallas en la comunicación.
- En caso de que hayan k < n procesos que dejen de funcionar, se puede resolver el consenso en O((k+1) . n^2) mensajes.
- Si los procesos no son confiables, se puede resolver consenso ssi n>3.k y la conectividad es mayor que 2.k

Clusters

- Desde el punto de vista científico: conjunto de computadoras conectadas por una red de alta velocidad, con un scheduler de trabajos en común.
- Desde el punto de vista común: conjunto de computadoras que trabajan cooperativamente desde alguna perspectiva.
- Grids: conjunto de clusters, cada uno bajo un dominio administrativo distinto.
- Clouds: clusters en donde uno puede alquilar una capacidad fija o variable bajo demanda.

Scheduling en Sistemas Distribuídos

- Local: dar el procesador a un proceso.
- Global: asignar un proceso a un procesador (mapping).
  - Estático: en el momento de la creación del proceso (**afinidad**)
  - Dinámico: la asignación puede variar durante la ejecución (**migración**)
    - Iniciada por el procesador sobrecargado (**Sender Initiated**)
    - Iniciada por el procesador libre (**Receiver Initiated / Work Stealing**)
  - Compartir vs Balancear:
    - Compartir: repartir.
    - Balancear: repartir equitativamente.

Factores de una Política de Scheduling

- Transferencia: cuándo hay que migrar un proceso.
- Selección: qué proceso hay que migrar.
- Ubicación: a dónde hay que enviar el proceso.
- Información: cómo se difunde el estado.

Protección y Seguridad

- Protección
  - Conjunto de medidas y mecanismos.
  - Restricción: previene que usuarios accedan a recursos o datos de otros.
  - Control: define qué usuario puede hacer cada cosa.
- Seguridad:
  - Asegurarse de la autenticidad de los usuarios.
  - Impedir la destrucción o adulteración de los datos.

Seguridad de la Información

Preservación de las siguientes características: *Confidencialidad* Integridad * Disponibilidad

Seguridad de la Información != Seguridad Informática

Características de un Sistema de Seguridad

Los sistemas de seguridad suelen tener: *Sujetos: el sujeto más común del SO es el del usuario. Pueden ejecutar acciones, y a veces ser dueños de objetos.* Objetos: archivos, procesos, memoria, conexiones, puertos, etcétera. * Acciones: leer, escribir, copiar, abrir, borrar, imprimir, ejecutar, matar, etcétera.

La idea es poder definir qué sujetos pueden realizar qué acciones sobre qué objetos. Un sujeto puede ser a la vez un objeto.

- Usuarios: es un sujeto básico del SO.
- Grupos: colecciones de usuarios.
- Roles: conjunto de acciones que se asignan a usuarios o grupos, para definir qué pueden (o no hacer).
    - Ejemplo de roles: operador, usuario común, administrador.

Propiedades de un Sistema de Seguridad

- Autenticación: verificar identidad/veracidad
    - Proporcionar algo que sé, algo que tengo, algo que soy.
    - Varios factores de autenticación.
        - Contraseñas
        - Medios biométricos
        - Tokens
    - Fuerte uso de criptografía
- Autorización: definir permisos / capacidades
- Auditoría o Contabilidad: registrar todas las acciones que fueron llevadas a cabo en el sistema, sobre qué objetos, y por qué sujetos.

Criptografía

```
Rama de las matemáticas y de la informática que se ocupa de cifrar/descifrar la i
```

- El criptoanálisis es el estudio de los métodos que se utilizan para quebrar textos cifrados con objeto de recuprar la información original en ausencia de la clave.

Algoritmos de Encriptación Simétricos

```
Utilizan la misma clave para encriptar y desencriptar.
```

- Caesar
- DES
- Blowfish
- AES

Algoritmos de Encriptación Asimétricos

```
Utilizan claves distintas para encriptar y desencriptar. Fueron un gran avance c
```

- RSA

Algoritmos de Hash One-Way

```
No son algoritmos de encriptación. No permiten revertir el resultado. Presentan
```

- MD5
- SHA1
- SHA-256

Funciones de Hash

- En criptografía se utilizan hashes de una vía.
- La idea es que sea prácticamente imposible obtener la preimagen.
- Requisitos:
  - Resistencia a la preimagen. Dado h, debería ser difícil encontrar m tal que h=hash(m).
  - Resistencia a la segunda preimagen. Dado m1, debería ser difícil encontrar m2!=m1 tal que hash(m1)=hash(m2).
  - Etc.
- Muy útiles para almacenar contraseñas para que no se puedan leer.
- A la hora de hashear una clave, es muy importante usar un SALT y varias iteraciones, para dificultar revertir el hash, ya que las funciones de hash son muy rápidas.

Método RSA

- Autores: Ronald Rivest, Adi Shamir, Len Adleman.

- Se toman dos números de muchos dígitos.
- A uno se lo denomina clave pública, al otro clave privada.
- Cada persona necesita ambas:
    - Su clave privada (que protege).
    - Su clave pública (que difunde).
- Para cifrar un mensaje, el emisor interpreta cada letra como si fuera un número, y luego hace una cuenta que involucra la clave pública del receptor.
- Para descifrar un mensaje, el necesario aplica una cuenta con su clave privada.

Explicación RSA

- Tomar `p` y `q` primos
- Multiplicarlos: `n = p.q`
- Calcular: `n' = (p-1)(q-1)`
- Elegir un entero `2 < e < n'-1` que sea coprimo con `n'`.
- `e` y `n` van a ser la clave pública.
- Computar `d` para que cumpla que el resto de `d.e/n' = 1` ssi `d.e = n'` ssi `d = n'/e`.
- `d` y `n` van a ser la clave privada.
- Encriptar: calcular x = m^e (n), el resto de dividir m^e por n.
- Desencriptar: calcular y = c^d (n), el resto de dividir c^d por n.
- Este método sirve porque **factorizar es muy difícil (NP)**.

Firma digital con RSA

- Calcular el hash de un documento.
- Encriptar el hash con la clave privada.
- Entregar el documento más el hash encriptado.
- El receptor lo desencripta con la clave pública. Luego calcula el hash del documento. si el hash desencriptado coincide con el del documento, entonces se asegura de que el autor del documento es quien dice ser.
- Todo esto está regulado mediante la Ley 25.506.

Autenticación remota con hash

- Existe un ataque llamado **"replay-attack"**, las funciones de hash no lo impiden.

- Los métodos que se utilizan para prevenir los replay-attack se llaman **Challenge-Response**.
  - El servidor elige un número al azar, y se lo envía al cliente.
  - El cliente tiene que encriptar la contraseña utilizando ese número como input.
  - El servidor hace lo mismo, y se fija si coinciden.
  - Lamentablemente, este método tampoco es infalible.

Autorización: Permisos

- Una forma de ver a los permisos es como una matriz de control de accesos.
  - Matriz de `Sujetos x Objetos`.
  - En cada celda figuran las acciones permitidas.
  - Se puede almacenar como una matriz centralizada, o separada por filas y columnas.
  - Los archivos suelen guardar qué puede hacer cada usuario con ellos.
- **Principio del Mínimo Privilegio**: se asume que todo lo que no está especificado, está prohibido.
  - Al crear un objeto nuevo, se le definen permisos por defecto.
  - Estos permisos por lo general están predefinidos según el tipo de objeto.

DAC vs MAC

- Discretionary Access Control: los atributos de seguridad se tienen que definir explícitamente, el dueño de cada objeto decide los permisos.
- Mandatory Access Control: cada sujeto tiene un grado, los objetos heredan el grado del último sujeto que los modificó, y un sujeto sólo puede acceder a objetos de grado menor o igual que el de él.
  - Modelo Bell-Lapadula.
  - Se lo utiliza para manejar información altamente sensible.

DAC en Unix

- SETUID y SETGID: permisos especiales que pueden asignarse a archivos o directorios en un sistema operativo Unix. Se utilizan principalmente para permitir a usuarios ejecutar binarios con los permisos de otro usuario, para realizar una tarea específica.
  - Si un archivo tiene el bit de SETUID activo, se identifica con una "S" al hacer `ls`, de la siguiente forma: `-rwsr-xr-x 1 root shadow 27920 ago 15 22:45 /usr/bin/passwd`
- Sticky Bit: si se activa sobre un directorio, todos los archivos contenidos en el mismo sólamente pueden ser borrados o renombrados por el dueño del archivo (o el dueño del directorio, o root). Se podría decir que cada archivo se queda "pegado" a su dueño.
- `chattr`: utilidad para cambiar atributos.

- Posix ACLs (getfacl, setfacl) y NFSv4 ACLs: son formas de ACL que se complementan y flexibilizan la forma estándar de unix, de forma que es posible definir permisos con mayor granularidad (ejemplo: a usuarios específicos, a más de un grupo, etcétera).

Puntos importantes

- Permiso para propagar o definir permisos.
- Revocación de permisos: ¿la aplicación debe ser inmediata o diferida?
  - Inmediata: puede causar un overhead considerable, ejemplo: cada vez que se cambia un permiso a uno o más archivos hay que verificar si esos descriptores se encuentran abiertos por algún usuario o proceso.
  - Diferida: puede ser origen de vulnerabilidades; ejemplo: si un usuario tiene un archivo abierto, con permisos de escritura, y se le bloquea ese permiso con posterioridad, el usuario va a poder seguir escribiendo a ese archivo.

Sistemas Distribuidos

Modelo de Fallas

- Nadie falla.
- Los procesos caen pero no se levantan.
- Los procesos caen y se levantan.
- Los procesos caen y se levantan pero sólo en determinados momentos.
- La red se particiona.
- Los procesos pueden comportarse de manera impredecible.

Cada modelo induce algoritmos distintos.

Métricas

- Cantidad de mensajes que se envían a través de la red.
- Tipos de fallas que soportan.
- Cuánta información requieren.
  - El tamaño de la red.
  - La cantidad de procesos.
  - Cómo ubicar a cada uno de ellos.

Problemas

- Orden de ocurrencia de los eventos

- Exclusión mutua
- Consenso

Exclusión mutua distribuida

- La forma más sencilla se llama **token passing**
  - La idea es armar un anillo lógico entre los procesos y poner a circular un token.
  - Cuando quiero entrar a la sección crítica espero a que me llegue el token.
  - *Si no hay fallas, no hay inanición*.
  - Siempre hay mensajes circulando aún cuando no son necesarios.
  - Fiber Distributed Data Interface (FDDI)
  - Time-Division Multiple-Access (TDMA)
  - Timed-Triggered Architecture (TTA)
- Otra forma
  - **No circulan mensajes si no se quiere entrar a la sección crítica.**
  - Cuando quiero entrar a la sección crítica envío `solicitud(Pi, ts)`, siendo ts el timestamp.
  - Cada proceso puede responder inmediatamente, o encolar la respuesta.
  - Sólamente puedo entrar a la sección crítica cuando recibí todas las respuestas.
  - Al salir, respondo a todos los pedidos pendientes.
  - Respondo inmediatamente si:
    - No quiero entrar a la sección crítica.
    - Si quiero entrar, aún no lo hice, y el ts del pedido que recibo es menor que el mío.
  - Este algoritmo exige que todos conozcan la existencia de todos.
- **Se asume que:**
  - No se pierden mensajes
  - Ningún proceso falla

Locks Distribuídos

- Procolo de Mayoría
  - Queremos obtener un lock sobre un objeto del cual hay copia en `n` lugares.
  - Para obtener un lock, debemos pedirlo a por lo menos `n/2+1` sitios.
  - Cada sitio responde si puede o no dárnoslo.
  - Cada copia del bojeto tiene un número de versión. Si lo escribimos, tomamos el más alto, y lo incrementamos en uno.
  - **Puede producir deadlock, hay que adaptar los algoritmos de detección.**

    - No es posible otorgar dos locks a la vez, ya que para ello cada proceso debería tener al menos `n/2+1` locks. Es decir que habrían `2(n/2+1)=2n+2` locks, lo cual es imposible.
    - Tampoco es posible leer una copia desactualizada, ya que para que eso suceda deberían haber `k>=n/2+1` locks cuya máxima marca sea `t` y existir otra copia `j` cuya marca `t_j > t`. Eso significiaría (por contradicción) que el último que escribió la versión `t_j`, lo hizo en menos de `n/2+1` copias, ya que si lo hubiera hecho en más de `n/2+1` copias, habrían en total más de `2n+2` copias, al menos `n/2+1` con la versión desactualizada, y al menos `n/2+1` con la versión actualizada.

Elección de Líder

```
Problema: Una serie de procesos debe elegir a uno como líder para la ejecución de
```

- En una red sin fallas, es sencillo.
    - Le Lann, Chang y Roberts (N. Lynch, cap 3 y cap 15.1)
    - Mantengo un status que dice que no soy el líder.
    - Organizo los procesos en anillo, hago circular mi ID.
    - Cuando recibo un mensaje, comparo el ID que circula con el mío. Hago circular el mayor.
    - Cuando el mensaje dio toda una vuelta, todos los procesos saben quién es el lider.
    - Ponemos a girar un mensaje de notificación para que todos lo sepan.
- Complicaciones posibles:
    - Varias elecciones simultáneas.
    - Procesos que suben y bajan del anillo.
- Tiempo:
    - Sin fase de stop O(n)
    - Con fase de stop O(2n)
- Comunicación:
    - $O(N^2)$
    - Cota inferior $\Omega(n \log n)$.
        - Algoritmo de Hirshberg y Sinclair.

Instantánea Global Consistente

- Sea un estado `E = ΣEi`, siendo Ei la parte del estado que le corresponde al proceso Pi.
- Lo único que modifica el estado son los mensajes que los procesos se mandan entre sí, y no los eventos externos.
- Quiero obtener una **instantánea (snapshot)** consistente de E.

- Esto es, en un momento dado, a partir de que hago el pedido, cuánto valían los Ei y qué mensajes había circulando en la red.
- Para ello, se envía un mensaje de *marca*.
- Cuando Pi recibe un mensaje de marca por primera vez, guarda una copia Ci = Ei, y envía un mesnaje de marca a todos los otros procesos.
- Luego, Pi empieza a registrar todos los mensajes que recibe de cada vecino Pj, hasta que recibe marca de todos ellos.
- En ese momento queda conformada la escuencia Recibidos(i,j) de todos los mensajes que recibió Pi de Pj antes de que éste tomara la instantánea.
- El estado global es que cada proceso está en el estado Ci y los mensajes que están en Recibidos están circulando por la red.

== TODO: Repasar esto de https://en.wikipedia.org/wiki/Chandy-Lamport_algorithm ==

Two-Phase Commit

- La idea es realizar una transacción de manera atómica. Todos los nodos deben estar de acuerdo en qué se hizo y qué no se hizo.
- En la primera fase se le preguntan a todos si están de acuerdo con realizar la transacción.
  - Si se recibe un no, se aborta la transacción.
  - Se van anotando todos los sí que se reciben, hasta tenerlos todos.
  - Si pasado un tiempo no se recibieron todos los sí, también se aborta.
- Si se reciben todos los sí, se avisa a todos que la transacción quedó confirmada.
- No protege contra todas las fallas, pero sí contra muchas.
- Terminación débil: si no hay fallas, todo proceso decide.
- Terminación fuerte: todo proceso que no falla, decide.
- **Teorema**: Two-phase commit resuelve COMMIT con terminación débil.
  - Two-phase commit no satisface terminación fuerte.
  - La solución es three-phase commit (N. Lynch, cap 7.2 y 7.3)

Consenso: acuerdos y aplicaciones

- Acuerdos:
  - k-agreement (o k-set agreement): decide(i) pertenece a W, tal que `|W| = k`
  - Aproximado: para todo i!=j, `|decide(i)-decide(j)| <= e`
  - Probabilístico: `Pr[existe i!=j, decide(i)!=decide(j)] < e`
- Aplicaciones:

- Sincronización de relojes (NTP, RFC 5905 y anteriores)
- Tolerancia a fallas en sistemas críticos

Bibliografía Adicional

- Nancy Lynch, Distributed Algorithms, Morgan Kaufmann, 1996. ISBN 1-55860-348-4.
- Hermann Kopetz, Gunther Bauer: The time-triggered architecture. Proceedings of the IEEE 91(1): 112-126 (2003). http://goo.gl/RPqfas
- R. Jain. FDDI Handbook. Addison Wesley, 1994. http://goo.gl/YZ2Hyl

Microkernels

Ventajas vs Kernel Monolítico

- Menos código privilegiado
- Facilidad de actualizar
- Mayor flexibilidad y extensibilidad
- La caída de un servicio no tira abajo el sistema
- Diferentes "sabores" de los servicios

Idea Inicial

Para lograr esto la idea era que tener un kernel que hiciera: *Manejo básico de memoria* Manejo básico de E/S * Manejo básico de IPC

El resto de las cosas serían provistas por servicios.

En la práctica...

- Resultó más lento que los kernels monolíticos.
- Hubo una segunda generación de microkernels que buscó solucionar estos problemas.
- La idea nunca terminó de ser exitosa.
- Excepciones:
  - QNX, un Unix RT con arquitectura microkernel, diseñado para sistemas embebidos.
  - MacOS tiene código de Mach, un microkernel. Pero el código que tiene es de cuando Mach no era un microkernel... así que técnicamente MacOS no usa microkernel.

Ideas útiles

Algunas ideas se salvaron.

- IPC más rápido.
- **Módulos de Kernel (!)**
- Intentar sacar *algunos servicios* del Kernel.

Esto significa que hoy en día los Kernel se pueden considerar híbridos.

Virtualización

```
Es la posibilidad de que un conjunto de recursos físicos sean vistos como varios
Coloquialmente, una computadora ejecutando varias computadoras.
```

- Máquinas virtuales, concepto que viene desde hace rato (al menos 1960).
  - Portabilidad (ej JVM)
  - Simulación
  - Testing
  - Aislamiento (chroot, jail, containers)
  - Particionamiento de Hardware
  - Agrupamiento de funciones (consolidation)
  - Protección ante fallas

Conceptos:

- VMM/Hypervisor: es un software/firmware/hardware específico destinado a crear, correr y administrar máquinas virtuales.
  - Tipo 0 (Hardware / Firmware): Es un tipo especial de virtualización, en donde el propio hardware se encarga de la virtualización, generando "particiones" o "dominios" de ejecución.
    - Cualquier Sistema Operativo que corra en un dominio, ve el hardware como si fuera exclusivo, no sabe que en realidad posiblemente está siendo compartido con otros Sistemas Operativos en otros dominios.
    - Funciona de forma "transparente".
    - La administración de IO puede generar problemas.
  - Tipo 1 (Nativo / Unhosted / Bare Metal): Se ejecuta directamente sobre el hardware.
    - Si bien se ejecuta directamente sobr el hardware, el VMM en sí es un Sistema Operativo, que provee funciones mínimas e indispensables (CPU scheduling,

administración de IO, protección y seguridad).
        - Muchas veces se provee, además, una API de administración (local, remota).
        - Ejemplos: VMware ESXi, XEN, Citrix XenServer, Microsoft Hyper-V Server, Oracle VM.
    - Tipo 2 (Hosteado): Se ejecuta como un software más.
        - Técnicamente, puede correr sobre cualquier Sistema Operativo, sin requerir de un soporte especial.
        - Por lo general la mayoría de las opciones requieren privilegios de Kernel, y para ello usan módulos específicos en el sistema operativo Host.
        - Todo esto obviamente agrega un overhead a la virtualización.
        - Ejemplos: Virtualbox, VMware Workstation/Server/Player, QEMU, Virtual PC.
    - Paravirtualización: No es exactamente virtualización. Requiere de un host y un guest modificados especialmente para funcionar.
        - Traduce las llamadas al sistema operativo guest en llamadas al sistema operativo host, por lo que estas corren localmente en el host.
        - El sistema operativo guest tiene que estar adaptado para utilizar la para-API provista por el VMM, por lo que un sistema operativo convencional (sin modificar) no puede correr sobre un VMM con paravirtualización.
        - Esta técnica tiene muchas ventajas en cuanto a performance.
        - Caso típico: Xen
    - Virtualización del Entorno de Programación: es virtualización basada en un modelo de ejecución distinto al típico, en donde lo que se virtualiza es el entorno de ejecución. Requiere contar con un soporte de Lenguaje/Software.
        - Caso típico: JVM
- Características esenciales de los VMM (Popek and Goldberg virtualization requirements)
    - Fidelidad (equivalence): Un programa corriendo en una VMM debe exhibir un comportamiento esencialmente idéntico al que tendría si estuviese corriendo en una máquina física.
    - Performance (efficiency): Una porción estadísticamente significativa de las instrucciones de máquina deben ser ser ejecutadas sin la intervención del VMM (es decir, para que no sea emulación).
    - Control de Recursos (safety): El VMM debe estar en completo control de los recursos virtualizados.

Simulación

- En el sistema *Host/Anfitrión* se construye una estructura de estado artificial que representa al sistema huésped.
- Se lee cada instrucción, y se modifica el estado, como si la instrucción se estuviese ejecutando realmente.

- **Problemas:**
    - Este mecanismo puede ser muy lento.
    - Hay cosas difíciles de simular, como DMA, concurrencia, etc.

Emulación de Hardware

- Se emulan los componentes de Hardware.
    - Cuando una máquina cree que está haciendo E/S, en realidad se está comunicando con el controlador de máquinas virtuales.
    - Este a su vez hace de proxy, ya sea usando el dispositivo real como backend, o el sistema de emulación que esté utilizando (puede ser emulación pura).
- La mayor parte del código se corre mediante traducción binaria.

- **Problemas:**
    - ¿Cómo logro separación de privilegios dentro de la máquina virtual? Toda la máquina virtual corre en modo usuario.
    - Overhead en el acceso a los dispositivos. Más lento.

Virtualización asistida por Hardware

```
Extensiones de arquitectura de procesador, que buscan facilitar la tarea de virtu
```

- Busca evitar los problemas descriptos anteriormente.
    - Ring aliasing: es el problema que surge de correr programas que fueron hechos para correr en un determinado anillo de protección, pero terminan corriendo en otro. Por ejemplo, programas pensados para correr en modo kernel, que terminan corriendo en modo usuario. Esto trae problemas, por ejemplo, de permisos al intentar ejecutar instrucciones privilegiadas.
    - Address-space compression: proteger el espacio de memoria del VMM, pero sin restringirla totalmente; es decir, se debe permitir el acceso (no arbitrario) del guest a dicha memoria. Por otro lado, para el Sistema Operativo Host, el VMM y el guest son el mismo proceso, por lo que la protección de memoria no se puede dar por ese lado.

    - Non-faulting access to privileged state: la protección basada en privilegios impide que el software no privilegiado acceda a determinados componentes del estado del CPU (necesarios para correr un Sistema Operativo). Algunas instrucciones privilegiadas generan un trap cuando se ejecutan sin permiso, y eso es bueno ya que se pueden capturar y resolver. Pero hay otras que no.
    - Interrupt virtualization: es necesario proveer soporte a interrupciones externas.
    - Access to hidden state: algunas cuestiones del estado del procesador no son accesibles por software.
    - Ring compression: tanto el kernel huésped como sus programas corren en el mismo nivel de privilegio; esto significa que no hay protección en el sistema operativo guest.
    - Frequent access to privileged resources: bloquear el acceso a cada recurso mediante un TRAP puede generar un overhead y un cuello de botella importante para recursos accedidos con frecuencia.

- En el caso de Intel, se agregaron las extensiones VT-x
    - Se agrega un nuevo anillo de privilegios (-1).
    - Se proveen dos modos:
        - VMX root: Las instrucciones se comportan de forma similar, pero con algunas extensiones (modo anfitrión).
        - VMX non-root: El mismo set de instrucciones, pero con comportamiento restringido (modo huésped).
    - Se proveen 10 instrucciones que permiten alternan entre modos.
    - Se agrega la VMCS (Virtual Machine Control Structure).
        - Campos de control: indica qué interrupciones recibe el huésped, qué puertos de E/S, etc.
        - Estado completo del huésped.
        - Estado completo del anfitrión.
    - La idea es que el CPU salga automáticamente del modo VMX non-root cuando el huésped realice alguna acción "prohibida" de acuerdo a la VMCS.
        - En ese momento el controlador de VM recibe el control, y emula, ignora o termina la acción prohibida.

Desafíos / Problemas

- Los sistemas operativos tienen optimizaciones que suponen que están en una máquina real.
    - Por ejemplo, el Kernel y el FS están optimizados para acceder al disco de forma eficiente.
    - La administración de memoria y scheduler de CPU suponen que la carga que ven es la real del hardware. Si no cuentan con esa información, pueden terminar tomando

decisiones que impacten en la performance del sistema.
- Contención: al estar todo en una misma máquina física, hay un único punto de falla.

Escenarios de uso / Ventajas

- Correr sistemas viejos o legacy.
- Aprovechamiento del equipamiento.
- Desarrollo / Testing / Debugging.
- Abaratar costos.
  - Maximizar la utilización del HW.

Bibliografía

- http://www.cs.cornell.edu/home/ulfar/ukernel/ukernel.html
- “Formal Requirements for Virtualizable Third Generation Architectures”, de Popek y Goldberg.
- “Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization”. http://www.intel.com/technology/itj/2006/v10i3/1-hardware/1-abstract.htm
- “A comparison of software and hardware techniques for x86 virtualization”, de Adams y Agesen.