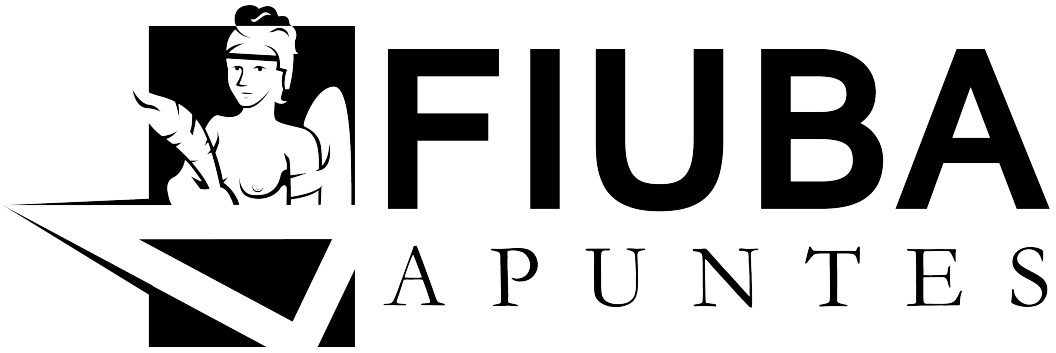


Sistemas Operativos (75.08 - 95.03)

Resumen teórico

fiuba-apuntes.github.io

Última actualización: 08/03/2015

LICENCIA

Este es un resumen (y no un sustituto) de la licencia. Este resumen destaca sólo algunas de las características clave y los términos de la licencia real. No es una licencia y no tiene valor legal. Usted debe revisar cuidadosamente todos los términos y condiciones de la licencia actual antes de usar el material licenciado.

Usted es libre para:

Compartir — copiar y redistribuir el material en cualquier medio o formato

Adaptar — remezclar, transformar y crear a partir del material

El licenciante no puede revocar estas libertades en tanto usted siga los términos de la licencia

Bajo los siguientes términos:

Atribución — Debe reconocer adecuadamente la autoría, proporcionar un enlace a la licencia e indicar si se han realizado cambios. Puede hacerlo de cualquier manera razonable, pero no de una manera que sugiera que tiene el apoyo del licenciador o lo recibe por el uso que hace.

NoComercial — Usted no puede hacer uso del material con fines comerciales.

CompartirIgual — Si usted mezcla, transforma o crea nuevo material a partir de esta obra, usted podrá distribuir su contribución siempre que utilice la misma licencia que la obra original.

No hay restricciones adicionales — Usted no puede aplicar términos legales ni medidas tecnológicas que restrinjan legalmente a otros hacer cualquier uso permitido por la licencia.

Aviso:

Usted no tiene que cumplir con la licencia para los materiales en el dominio público o cuando su uso esté permitido por una excepción o limitación aplicable.

No se entregan garantías. La licencia podría no entregarle todos los permisos que necesita para el uso que tenga previsto. Por ejemplo, otros derechos como relativos a publicidad, privacidad, o derechos morales pueden limitar la forma en que utilice el material.

Índice

Acerca del proyecto

FIUBA Apuntes nació con el objetivo de ofrecer en formato digital los apuntes de las materias que andan rondando por los pasillos de FIUBA y que los mismos sean fácilmente corregidos y actualizados.

Cualquier persona es libre de usarlos, corregirlos y mejorarlos.

Encontrarás más información acerca del proyecto o más apuntes en fiuba-apuntes.github.io.

¿Por qué usamos LaTeX?

LaTeX es un sistema de composición de textos que genera documentos con alta calidad tipográfica, posibilidad de representación de ecuaciones y fórmulas matemáticas. Su enfoque es centrarse exclusivamente en el contenido sin tener que preocuparse demasiado en el formato.

LaTeX es libre, por lo que existen multitud de utilidades y herramientas para su uso, se dispone de mucha documentación que ayuda al enriquecimiento del estilo final del documento sin demasiado esfuerzo.

Esta herramienta es muy utilizado en el ámbito científico, para la publicación de papers, tesis u otros documentos. Incluso, en FIUBA, es utilizado para crear los enunciados de exámenes y apuntes oficiales de algunos cursos.

¿Por qué usamos Git?

Git es un software de control de versiones de archivos de código fuente desde el cual cualquiera puede obtener una copia de un repositorio, poder realizar aportes tanto realizando *commits* o como realizando *forks* para ser unidos al repositorio principal.

Su uso es relativamente sencillo y su filosofía colaborativa permite que se sumen colaboradores a un proyecto fácilmente.

GitHub es una plataforma que, además de ofrecer los repositorios git, ofrece funcionalidades adicionales muy interesantes como gestor de reporte de errores.

1. Introducción

1.1. ¿Qué es un sistema operativo?

- Un programa que hace de intermediario entre el usuario de la computadora y su hardware (Oculta los detalles finos de la arquitectura).
- Un programa que administra los recursos de un sistema de computación: permite administrar el tiempo de procesador y el espacio (memoria, disco, etc).

1.2. Arquitecturas

1.2.1. Mainframe

Computadora central. Gran capacidad de I/O, server para e-commerce a gran escala.

Seguridad y disponibilidad:

- Transaction processing: es procesamiento de información distribuido en operaciones individuales e indivisibles, llamadas *transacciones*. Cada transacción debe ser exitosa o fallar como unidad entera, no puede haber transacciones parcialmente completas.
- Batch processing: Es la ejecución de una serie de programas ("tareas") en una computadora sin intervención del usuario.

1.2.2. Servidores

Destinados a ofrecer servicios a través de una red.

1.2.3. Supercomputadoras

Computacion de alto rendimiento. Se usan para hacer simulaciones.

Limites:

- Concurrencia: los procesos no son 100 % independientes
- Costo
- Programación del software

1.2.4. Server operating system

Interfaz solo línea de comando o EFI (estándar de firmware).

1.2.5. Computadora personal

No requiere conocimientos especiales.

1.2.6. Tablets, PDA

1.2.7. Consolas

1.2.8. Sistemas operativos embebidos

Dispositivos que no aceptan instalación de nuevo software por el usuario.
No deberían tener bugs.
Se usan en tvs, autos, etc.

1.2.9. Cluster

Un grupo de computadoras interconectadas por una red local de alta velocidad.
Se comportan como si fuese una única computadora.
Si es de alta disponibilidad tiene nodos redundantes en caso de falla. Retoma en otro equipo en el estado en el que estaba.
Balance de carga, con dispositivo físico o de software.

1.2.10. Grid

Cluster virtual con recursos distribuidos.
Ejemplo: BOINC, SETI.
Problemas: concurrencia (que se choquen tareas), que queden tareas sin cubrir.

1.2.11. Cloud computing

Se provee por internet. Dinamicamente escalable.
Atrás de la nube puede haber cluster, grid, etc (al cliente no le importa).
Servicios posibles de cloud computing:

- Cloud Storage: Dropbox.
- Infraestructura (infraestructura as a service (IaaS)):Tipicamente plataformas virtualizadas. Ejemplo: Amazon EC2
- Plataforma (PaaS): Provee la plataforma y un ambiente de desarrollo y soporte. Ejemplo: Google Code.
- Software (SaaS): Software on demand provisto por terceros. Ejemplo: Amazon Services, Paypal.

1.2.12. Tiempo real

Distinto de online o de rápido.
Tiempo de respuesta máximo y predecible.

1.2.13. Multiprocesador

Más de un procesador en el mismo chip o board.
Soportado en todos los sistemas operativos de escritorio.
La paralelizacion esta limitada por la ley de Amdahl: El *speedup* de un programa que utiliza varios procesadores en paralelo está limitado por el tiempo tomado por la fracción secuencial del programa.

2. Mecanismos básicos

Sistema operativo es software que extiende un poco la capa de hardware.

El hardware es lo que le provee recursos al sistema operativo: CPU, memoria, dispositivos I/O. Cada nivel interpreta al nivel superior.

El estado de una maquina virtual sólo está definido entre instrucción e instrucción.

Una instrucción en una capa equivale a muchas instrucciones de la capa inferior.

2.1. Modos de CPU

Son distintos niveles de permisos o privilegios.

Se suele trabajar con dos modos: **Modo Supervisor** (puede hacer todo) y **Modo Usuario** (tiene restricciones).

Se pasa de modo usuario a modo supervisor por medio de una interrupción.

El retorno a modo usuario está a cargo del programa. Motivo para querer pasar de modo supervisor a modo usuario: Control de riesgo de código desconocido (ejemplo: escribir en las direcciones de memoria del SO).

Algunas arquitecturas incluyen más modos:

- X86 Modo real, protegido y virtual.
- Modo hypervisor

Un sistema operativo puede tener partes corriendo en cada uno de los modos.

Un programa de usuario sólo corre en modo usuario. El único programa que debiera ser capaz de pasar a modo supervisor es el sistema operativo.

2.2. Interrupciones

Una interrupción es una suspensión temporal de la ejecución de un proceso, para pasar a ejecutar una subrutina de servicio de interrupción, la cual, por lo general, no forma parte del programa, sino que pertenece al sistema operativo o al BIOS. Una vez finalizada dicha subrutina, se reanuda la ejecución del programa.

Hay dos tipos de interrupciones:

- Sincronica o software trap: Una instrucción del programa.
- Asincrónica: I/O, timer, external.

2.2.1. Atención de interrupciones

1. **Primer nivel de atención:** Salvar el contexto (registros, código de condición, dirección de retorno). El objetivo es poder proseguir el proceso (después de atendida la interrupción) desde el estado en el que estaba. Que un proceso sufra o no una interrupción no cambia su resultado, solamente el tiempo que le insume.
2. **Segundo nivel de atención:** Se decide si se atiende en el momento la interrupción o se deja para después. Si vienen 2 interrupciones al mismo tiempo puede llegar a perderse una. Por eso los que envían la interrupción deben estar preparados para repetirla.

2.3. Modos del sistema operativo

2.3.1. Modo Kernel

Ejecutando un servicio propio del sistema operativo.

En computación, el *kernel* es un programa que maneja las solicitudes de entrada y salida que realiza el software, traduciendolas en instrucciones de procesamiento de datos para la CPU y otros componenentes electrónicos de una computadora. El kernel es una parte fundamental en un sistema operativo moderno de PC.

Debido a su naturaleza crítica, el código kernel generalmente se encuentra cargado en un area protegida de la memoria, previniendo que sea sobreescrito por otra parte no tan usada del sistema operativo o por aplicaciones. El kernel realiza sus tareas, como la ejecución de procesos y manejo de interrupciones, en area del kernel,

mientras que todo lo que un usuario normalmente haría, como escribir texto en un editor o correr aplicaciones con interfaz gráfica, se realiza en el espacio del usuario. Esta separación se realiza con el fin de prevenir datos del usuario y del kernel interferir uno con el otro, disminuyendo performance o causando el sistema operativo inestable (o incluso colgandolo)

Cuando un programa (o como se lo conoce en este contexto *proceso*) realiza solicitudes al kernel, esta solicitud se llama "llamada al sistema" o *system call*

2.3.2. Modo usuario

Ejecutando un programa de usuario.

The term userland (or user space) refers to all code which runs outside the operating system's kernel. Userland usually refers to the various programs and libraries that the operating system uses to interact with the kernel: software that performs input/output, manipulates file system objects, application software etc.

2.4. System Calls

Si un proceso esta corriendo un programa en modo usuario y necesita un servicio del sistema, como leer data de un archivo, tiene que ejecutar un software trap para transferirle el control al sistema operativo. El sistema operativo se fija lo que necesita el proceso que lo llamo inspeccionando los parámetros. Hace lo que tenga que hacer y devuelve el control a la instrucción que sigue al system call.

Ejemplos

- Leer de un dispositivo solo puede hacer el SO (leer de memoria, CD, USB, etc).
- Manejo de procesos
- Manejo de archivos
- Etc

2.5. Library Calls

Son llamados a procedimientos de bibliotecas provistas por el lenguaje en el que se está programando.

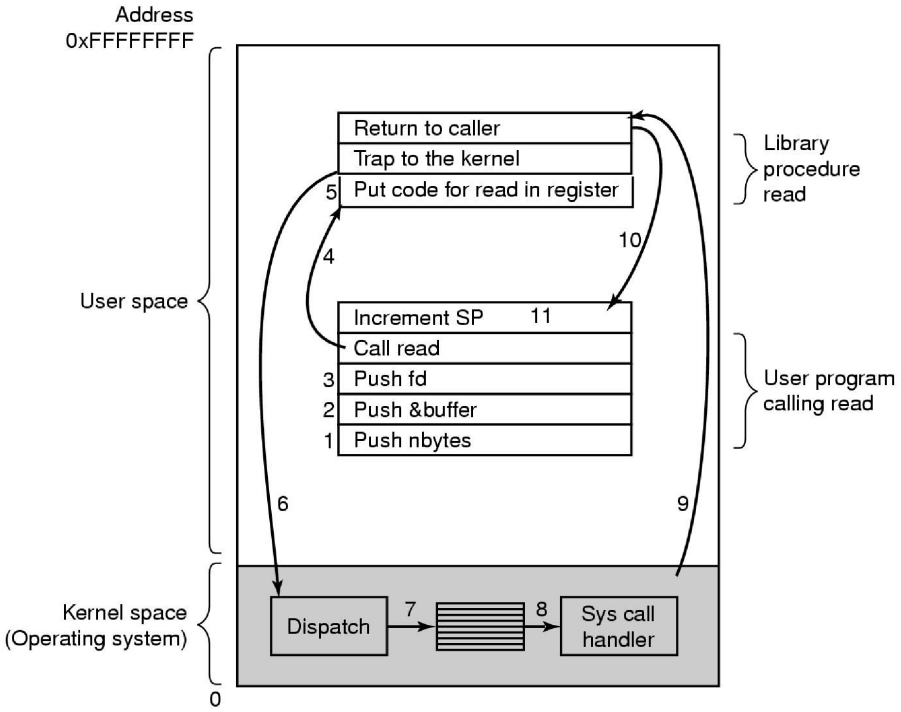


Figura 1: Llamado a procedimientos de bibliotecas

Referencias de la Figura 1:

- 1, 2, 3 y 4 corresponden a un llamado convencional a un procedimiento.
- Antes de hacer 5, el sistema operativo habilita el acceso a la memoria del sistema operativo (pasa a modo kernel).
- 6 implica una software trap, el procesador pasa a modo protegido.
- 7 y 8 se ejecutan en modo protegido del procesador.
- En 9 se vuelve a modo usuario del procesador, pero con el sistema operativo en modo kernel.
- El paso 10 reestablece la protección de memoria del sistema operativo (sale de modo kernel).

3. Procesos

3.1. Modelo de procesos

El sistema operativo debe organizar el software que corre en unidades secuenciales, a esta organización se le llama proceso.

Un proceso es:

- La imagen de un programa en ejecución (una copia del programa).
- Con las estructuras del sistema operativo para administrarlo.
- Varios procesos pueden estar asociados a un mismo programa; por ejemplo, iniciar varias instancias de un mismo programa generalmente significa que más de un proceso está siendo ejecutado.

Un proceso tiene:

- La imagen del programa (una copia de su código ejecutable y de su área de datos).
- La información acerca de sus estado de ejecución:
  - Los valores del program counter, registros y variables.
  - Información necesaria para su administración por parte del Sistema Operativo (id, prioridad, ...).
- Memoria (generalmente una región de memoria virtual); que incluye el código ejecutable, datos específicos del proceso (entrada y salida), un stack de llamadas (para mantener registro de las subrutinas activas y otros eventos), y un heap para mantener datos intermedios generados durante el tiempo de ejecución.
- Descriptores de recursos del sistema operativos, reservados por el proceso, como pueden ser los *file descriptors* (Unix), o *handles* (Windows), y fuentes y sumideros de datos.
- Atributos de seguridad, como el propietario del proceso y los permisos (operaciones permmitidas) del mismo.

Esta información la guarda el sistema operativo en estructuras de datos llamadas *Process Control Blocks*.

Cualquier subgrupo de recursos, menos, generalmente, el estado del procesador, puede estar asociado con cada uno de los threads del proceso (en sistemas operativos que soportan threads) o en los procesos "hijos".

El sistema operativo mantiene sus procesos separados y reserva los recursos que necesitan, de manera que sean menos propensos a interferir entre ellos y causen fallas del sistema (como deadlocks o thrashing). El sistema operativo también provees mecanismos para la comunicación entre procesos, permitiendoles interactuar de manera segura y predecible.

3.2. Multiprogramación

En computación, *multitasking* es un metodo donde multiples tareas (procesos) son ejecutadas durante el mismo periodo de tiempo. Se ejecutan *concurrentes* (en periodos de tiempo solapados, una tarea puede iniciar antes que otras hayan terminado) en vez de *secuenciales* (una tarea comienza luego de que la anterior haya terminado). Las tareas concurrentes comparten recursos de procesamiento, como la CPU y memoria principal.

Multitasking no necesariamente significa que varias tareas se ejecutan en el mismo preciso momento. En otras palabras, multitasking NO implica paralelismo, pero si significa que más de una tarea puede estar en medio de su ejecución al mismo tiempo, y que mas d euna tarea está avanzando dentro de un periodo de tiempo determinado.

En caso de una computadora con un solo CPU, solo una tarea se dice estar corriendo en determinado tiempo, lo que significa que ese CPU está activamente ejecutando instrucciones para esa tarea.
Multitasking resuelve este problema organizando qué tarea se ejecutará en cada tiempo determinado, y cuándo una tarea en espera obtiene un turno. El acto de reasignar un CPU de una tarea a otra se llama *context switch*, o cambio de contexto. Cuando estos cambios de contexto ocurren lo suficientemente seguidos, se logra una ilusión de paralelismo.

Incluso en computadoras con más de un CPU (máquinas *multiprocesadores*) o más de un nucleo en determinado CPU (máquinas *multinúcleo*), donde más de una tarea puede ser ejecutada en un determinado instante (una por núcleo), multitasking permite correr muchas más tareas que la cantidad de CPUs presentes.

Cuando hay más de un procesador se conoce como Multiprocesamiento.

Se ejecuta un proceso. Cuando se "bloquea" por I/O, se aprovecha el tiempo para ejecutar otro proceso.

La CPU va conmutando (switching) de un proceso a otro.

Es un multiplexado de la CPU.

3.2.1. Implementación de la multiprogramación

- Se conoce como *scheduler* al mecanismo que permite elegir varios procesos en estado *Ready*, para otorgarles tiempo de CPU y que puedan realizar sus tareas. Para ello aplica un algoritmo de scheduling, que tiene en cuenta los siguientes aspectos

  - Cantidad requerida de recursos.
  - Cantidad actualmente disponible de recursos.
  - Prioridad del trabajo o proceso.
  - la cantidad de tiempo de espera.

- *Dispatcher* es el mecanismo que otorga tiempo de CPU al proceso seleccionado por el scheduler. Para esto, se realizan los siguientes pasos:

  - Cambio de contexto (*Context switching*)
  - Cambiar a modo usuario (*user mode*)

  El tiempo se divide en segmentos, denominados *time slices*. Cuando un *time slice* se termina, le permite al scheduler actualizar el estado de cada proceso, y seleccionar el próximo a ejecutar.

Cada vez que se interrumpe un proceso también se pierde el tiempo de guardar el contexto.

3.3. Estados de un proceso

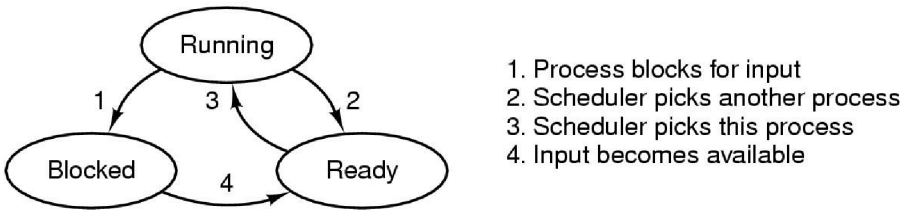


Figura 2: Estados de un procedimiento (simplificado)

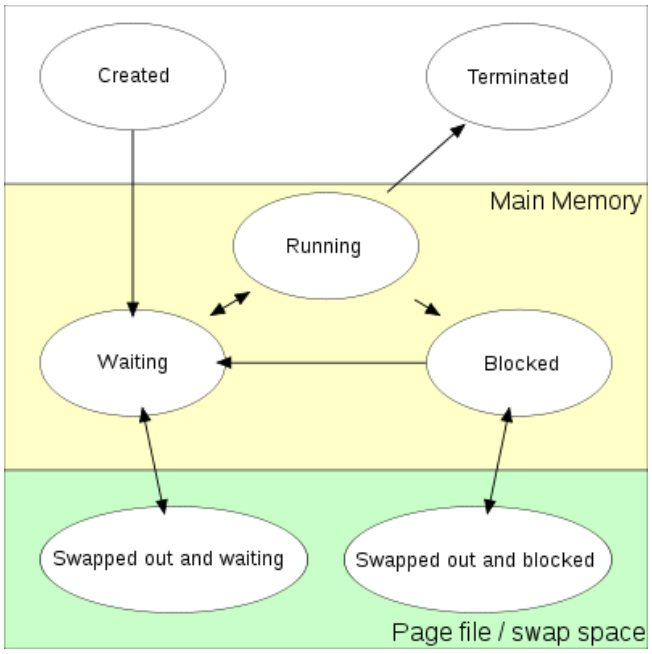


Figura 3: Estados de un procedimiento

3.3.1. Created or New

When a process is first created, it occupies the "created" or "new" state. In this state, the process awaits admission to the "ready" state. This admission will be approved or delayed by a long-term, or admission, scheduler. Typically in most desktop computer systems, this admission will be approved automatically, however for real-time operating systems this admission may be delayed. In a real time system, admitting too many processes to the "ready" state may lead to oversaturation and over contention for the systems resources, leading to an inability to meet process deadlines.

3.3.2. Ready and waiting

A "ready" or "waiting" process has been loaded into main memory and is awaiting execution on a CPU (to be context switched onto the CPU by the dispatcher, or short-term scheduler). There may be many "ready" processes at any one point of the system's execution—for example, in a one-processor system, only one process can be executing at any one time, and all other "concurrently executing" processes will be waiting for execution.

A ready queue or run queue is used in computer scheduling. Modern computers are capable of running many different programs or processes at the same time. However, the CPU is only capable of handling one process at a time. Processes that are ready for the CPU are kept in a queue for "ready" processes. Other processes that are waiting for an event to occur, such as loading information from a hard drive or waiting on an internet connection, are not in the ready queue.

3.3.3. Running

A process moves into the running state when it is chosen for execution. The process's instructions are executed by one of the CPUs (or cores) of the system. There is at most one running process per CPU or core. A process can run in either of the two modes, namely kernel mode or user mode.

3.3.4. Blocked (Waiting)

A process that is blocked on some event (such as I/O operation completion or a signal). A process may be blocked due to various reasons such as when a particular process has exhausted the CPU time allocated to it or it is waiting for an event to occur.

3.3.5. Terminated

A process may be terminated, either from the "running" state by completing its execution or by explicitly being killed. In either of these cases, the process moves to the "terminated" state. The underlying program is no longer executing, but the process remains in the process table as a zombie process until its parent process calls the wait system call to read its exit status, at which point the process is removed from the process table, finally ending the process's lifetime. If the parent fails to call wait, this continues to consume the process table entry (concretely the process identifier or PID), and causes a resource leak.

A child process always first becomes a zombie before being removed from the resource table. In most cases, under normal system operation zombies are immediately waited on by their parent and then reaped by the system – processes that stay zombies for a long time are generally an error and cause a resource leak.

Después de terminado un proceso, el mismo queda en estado "terminado" hasta que el sistema operativo termina de limpiar las estructuras que usó para ejecutarlo (mientras tanto está en estado zombie).

```
1  int main()
2  {
3      pid_t child_pid;
4
5      child_pid = fork();
6      if(child_pid > 0){
7          sleep(60);
8      }else{
9          exit(0);
10     }
11     return 0;
12 }
```

3.4. PCB (Process Control Block)

Es la estructura de datos con la que el sistema operativo administra los procesos. Contiene la información acerca del proceso y su estado. Además la información que el S.O. precisa para manejarlo como: Identificador, Estado, Recursos, Historia. Ejemplo de datos que maneja: Registros, Program counter, Process ID, Punteros de memoria, etc.

Estados de un proceso: Los estados se manejan como colas. El Dispatcher es el encargado de cambiar los PCBs entre las colas.

3.4.1. Dispatcher (Scheduler de corto plazo)

Decide qué proceso, entre los procesos en memoria y con estado *ready*, será ejecutado (otorgado un CPU) luego de una interrupción del clock, una interrupción de entrada/salida, llamada al sistema operativo u otro tipo de señal. Por este motivo, este scheduler de corto plazo realizar tareas de planificación mucho más seguido que los schedulers de medio y largo plazo. Al menos una decisión de scheduling se realiza luego de cada *time slice*, y estos son bastante cortos.

- Al pasar de Running a Blocked. El manejador de interrupciones lo invoca para cambiar de estado al proceso:
  - Salva los datos necesarios en el PCB.
  - Cambia el PCB de cola.

  Luego se decide a que proceso dar control (tarea del Scheduler).

- Al pasar de Ready a Running
  El Scheduler lo invoca cuando ya decidió a que proceso activar.
  Carga el estado de la CPU con los datos del PCB.
  Continúa la ejecución del proceso.

3.4.2. Scheduler (Long term)

Decide a cuál de los procesos en ready hay que darle el control. In general, most processes can be described as either I/O-bound or CPU-bound. An I/O-bound process is one that spends more of its time doing I/O than it spends doing computations. A CPU-bound process, in contrast, generates I/O requests infrequently, using more of its time doing computations. It is important that a long-term scheduler selects a good process mix of I/O-bound and CPU-bound processes. If all processes are I/O-bound, the ready queue will almost always be empty, and the short-term scheduler will have little to do. On the other hand, if all processes are CPU-bound, the I/O waiting queue will almost always be empty, devices will go unused, and again the system will be unbalanced. he system with the best performance will thus have a combination of CPU-bound and I/O-bound processes.

Tiene en cuenta las características del proceso:

- **Throughput** - The total number of processes that complete their execution per time unit.
- **Latency**, specifically:
  - **Turnaround time** - total time between submission of a process and its completion.
  - **Response time** - amount of time it takes from when a request was submitted until the first response is produced.
- **Fairness** - Equal CPU time to each process (or more generally appropriate times according to each process' priority and workload).
- **Waiting Time** - The time the process remains in the ready queue.

Objetivos del Scheduler:

- Dar una participación adecuada del reparto de tiempo de CPU (Fairness).
- Equilibrar el uso de recursos (Load Balancing).
- Aplicar las políticas generales del Sistema (prioridades, afinidad, seguridad).
- El resto depende del tipo de Sistema.

In practice, these goals often conflict (e.g. throughput versus latency), thus a scheduler will implement a suitable compromise. Preference is given to any one of the concerns mentioned above, depending upon the user's needs and objectives.

- Batch (por lotes):
  - maximizar el throughput: cantidad de procesos / tiempo.
  - Mantener la CPU ocupada.
  - Minimizar el turnaround time.
- Interactivo:
  - Buen tiempo de respuesta
  - Expectativas del usuario
- Real time:
  - Cumplir con los deadlines
  - Desempeño predecible

Las decisiones de scheduling se pueden tomar cuando un proceso:

- Pasa de running a blocked/waiting. (Transicion NO apropiativa)
- Pasa de running a ready. (Transicion apropiativa)
- Pasa de blocked/waiting a ready. (apropiativa)

- Termina. (NO apropiativa)

Transicion apropiativa: es el SO el que interrumpe.
Transcion no apropiativa: es el propio proceso el que interrumpe.
Starvation: es un problema que ocurre cuando hay multitasking, donde a un proceso se le niega constantemente los recursos necesarios. De esta manera la tarea nunca puede concretarse.

3.5. Algoritmos de scheduling

1. FIFO
2. Shortest Job Next (SJN)
3. Round Robin
4. Múltiples colas con prioridad

3.5.1. First come-First served (FIFO)

Simplemente encola procesos en estado *ready* en el orden de llegada.

Características

- Since context switches only occur upon process termination, and no reorganization of the process queue is required, scheduling overhead is minimal.
- Throughput can be low, since long processes can hold the CPU
- Turnaround time, waiting time and response time can be high for the same reasons above
- No prioritization occurs, thus this system has trouble meeting process deadlines.
- The lack of prioritization means that as long as every process eventually completes, there is no starvation. In an environment where some processes might not complete, there can be starvation.
- It is based on Queuing

3.5.2. Shortest Job Next

Selecciona para ser ejecutado el proceso en espera con el menor tiempo de ejecución.
Pros:

- Shortest job next is advantageous because of its simplicity and because it minimizes the average amount of time each process has to wait until its execution is complete.

Contras:

- However, it has the potential for process starvation for processes which will require a long time to complete if short processes are continually added.
- Another disadvantage of using shortest job next is that the total execution time of a job must be known before execution. While it is not possible to perfectly predict execution time, several methods can be used to estimate the execution time for a job, such as a weighted average of previous execution times.

3.5.3. Round Robin

Time slices are assigned to each process in equal portions and in circular order, handling all processes without priority (also known as cyclic executive).
Pros:

- Round-robin scheduling is simple, easy to implement, and starvation-free.
- Good average response time, waiting time is dependent on number of processes, and not average process length.

- Starvation can never occur, since no priority is given. Order of time unit allocation is based upon process arrival time, similar to FCFS.

Contras:

- Because of high waiting times, deadlines are rarely met in a pure RR system.

3.5.4. Múltiples colas con Prioridad

This is used for situations in which processes are easily divided into different groups. For example, a common division is made between foreground (interactive) processes and background (batch) processes. These two types of processes have different response-time requirements and so may have different scheduling needs. It is very useful for shared memory problems. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher priority queues. In addition, a process that waits too long in a lower-priority queue may be moved to a higher priority queue. This form of aging also helps to prevent starvation of certain lower priority processes.

3.6. Creacion/Terminacion de procesos

3.6.1. Creación de Procesos

- Al iniciar el sistema (Booting)
- Por pedido del usuario (Uso de una System Call).

3.6.2. Terminación de procesos

- Salida normal (voluntaria).
- Salida por error (voluntaria).
- Error “fatal” (involuntaria).
- “Muerte” por otro proceso.

3.7. Booting

A boot loader is a computer program that loads an operating system or some other system software for the computer after completion of the power-on self-tests; it is the loader for the operating system itself, which has its own loader for loading ordinary user programs and libraries. Within the hard reboot process, it runs after completion of the self-tests, then loads and runs the software. A boot loader is loaded into main memory from persistent memory, such as a hard disk drive or, in some older computers, from a medium such as punched cards, punched tape, or magnetic tape. The boot loader then loads and executes the processes that finalize the boot.

- Cargar en memoria un software que pueda lanzar un Sistema Operativo.
  - Switches en el panel.
  - Flash boot loader.
  - MBR (Master Boot Record) program.
  - EFI (Extended Firmware Interface).
- Termina cargando el first stage boot loader.

3.8. EFI (Extensible Firmware Interface)

Interfaz Extensible del Firmware, Extensible Firmware Interface (EFI), es una especificación desarrollada por Intel dirigida a reemplazar la antigua interfaz del estándar IBM PC ROM BIOS, e interactúa como puente entre el sistema operativo y el firmware base.

- Boot Services:
  - Soporte de consola.
  - Soporte gráfico.
- Runtime Services:
  - Device Drivers
  - Fecha y Hora
- Carga de código desde Internet

Las especificaciones de la EFI permiten ofrecer un controlador de dispositivo independiente del procesador denominado EFI Byte Code o simplemente EBC. Gracias a esto, se permite soporte para la carga de gráficos, red, sonido y opciones avanzadas del sistema, sin haber precargado el sistema operativo en cuestión. Esto era totalmente imposible en el BIOS, ya que cargaba funciones muy limitadas y necesarias como el soporte de periféricos como teclado y ratón.

3.9. UEFI

El 25 de julio de 2005 se creó la fundación UEFI (Unified Extensible Firmware Interface) cuya labor consistía en desarrollar y promocionar la plataforma EFI.

Define un "boot manager" (a firmware policy engine) que carga el loader del SO y los drivers que se necesiten. La configuración del booteo se almacena en variables NVRAM (path de loaders) Los loaders del SO son "clases" de aplicaciones UEFI, como clases que son, se almacenan en el file system (EFI System partition) que es independiente del medio (HD, Optical Disk, etc). Especifica un Shell para ejecutar aplicaciones (eje boot loaders), modificar variables, etc Mantiene compatibilidad reversa con BIOS

3.10. Proceso de BOOT – Linux

En Linux, el flujo de control durante el arranque es desde el BIOS, al gestor de arranque y al núcleo (kernel). El núcleo inicia el planificador (para permitir la multitarea) y ejecuta el primer espacio de usuario (es decir, fuera del espacio del núcleo) y el programa de inicialización (que establece el entorno de usuario y permite la interacción del usuario y el inicio de sesión), momento en el que el núcleo se inactiva hasta que sea llamado externamente.

3.11. Creacion de procesos por el usuario

Espacio de direcciones de un proceso

- TXT: Ejecutable
- DATA: Variables "static"
- U_Area: Stack + Información del proceso

3.11.1. TXT

The Text segment (a.k.a the Instruction segment) contains the executable program code and constant data. The text segment is marked by the operating system as read-only and can not be modified by the process. Multiple processes can share the same text segment. Processes share the text segment if a second copy of the program is to be executed concurrently.

3.11.2. Data

The data segment, which is contiguous (in a virtual sense) with the text segment, can be subdivided into initialized data (e.g. in C/C++, variables that are declared as static or are static by virtual of their placement) and uninitialized (or 0-initizliazed) data. The uninitialized data area is also called BSS (Block Started By Symbol). For example, Initialized Data section is for initialized global variables or static variables, and BSS is for uninitialized.

3.11.3. U_Area (STACK + Process info)

In addition to the text, data, and stack segment, the OS also maintains for each process a region called the u area (User Area). The u area contains information specific to the process (e.g. open files, current directory, signal action, accounting information) and a system stack segment for process use. If the process makes a system call (e.g., the system call to write in the function in main), the stack frame information for the system is stored in the system stack segment. Again, this information is kept by the OS in an area that the process doesn't normally have access to. Thus, if this information is needed, the process must use special system call to access it. Like the process itself, the contents of the u area for the process are paged in and out by the OS.

3.12. Fork y Exec

3.12.1. Exec

Reemplaza al proceso actual con un nuevo programa.

3.12.2. Fork

Lanza un nuevo proceso a imagen y semejanza de sí mismo. El hijo tiene el mismo código ejecutable que su padre. The fork operation creates a separate address space for the child When a process calls fork, it is deemed the parent process, and the newly created process, its child. After the fork, both processes not only run the same program, but they resume execution as though both had called the system call. They can then inspect the call's return value to determine their status, child or parent, and act accordingly. Cada uno tiene su propio espacio de direcciones. No se comparte memoria de escritura.

So, `fork()` and `exec()` are often used in sequence to get a new program running as a child of a current process. Shells typically do this whenever you try to run a program like `find` - the shell forks, then the child loads the `find` program into memory, setting up all command line arguments, standard I/O and so forth.

Laboratorio

```
if ( (pidhijo = fork()) == 0) {
    // When fork() returns 0, we are in the child process.
    cout << endl << "---> Es el HIJO con pid = " << getpid() << " cuyo padre es pid = " <<
        getppid() << endl;
    exit(0);
} else {
    // When fork() returns a positive number, we are in the parent process
    // and the return value is the PID of the newly created child process.
    cout << endl << "Es el PADRE con pid = " << getpid() << " y su hijo es pid = " << pidhijo
        << endl;
    exit(0);
}
```

Fork copia TXT, Data y U_Area "on demand"

```
if (fork()==0){
    Read    //lee el hijo
} else {
    Read    //lee el padre
}
```

Los dos leen del mismo archivo. Si leen una vez cada uno, cada uno lee salteado.

Lock compartido: Asociado al read. **Lock exclusivo**: Asociado al write.

Si quiero bloquear un archivo (de forma exclusiva) y ya esta bloqueado, se queda esperando hasta que se desbloquea.

El unico caso en que los bloquea al mismo tiempo es que ambos sean compartidos.

El bloqueo no es una cola. Lo hace o se queda esperando, sin orden de prioridad.
El lock es un protocolo o una convencion, pero no impide read, write, delete, etc, si se lo hace sin locks.

Ejecución Foreground con proceso hijo Unix

```
> script1.sh    #script1.sh necesita permiso de ejecución
# no nos devuelve el control hasta que no finaliza
```

Ejecución Background con proceso hijo

```
> script1.sh &      #script1.sh necesita permiso de ejecución
#Nos devuelve el control en el momento

[1] 20295       #muestra el número de proceso
[1] + Done script1.sh   #nos avisa que finalizó
```

Ejecución Foreground sin proceso hijo

```
> . .script1.sh         #script1.sh no necesita permiso de ejecución
#no nos devuelve el control hasta que no finaliza
#se ejecuta en el mismo ambiente, eso significa que
#no hay un shell hijo
```

4. Threads

Los thread son mini-procesos en el mismo espacio de direcciones que corren casi en paralelo. Como están en el mismo espacio de direcciones, comparten la data. Es decir, son hilos de ejecución que comparten el agrupamiento de recursos.

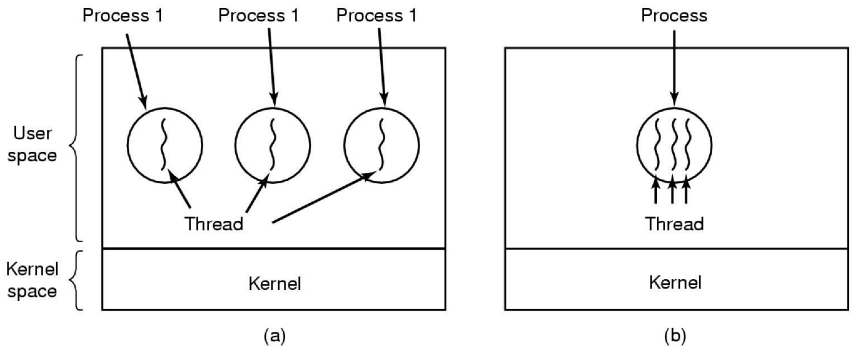
Son más livianos que los procesos, así es que es más fácil y rápido crearlos y destruirlos.

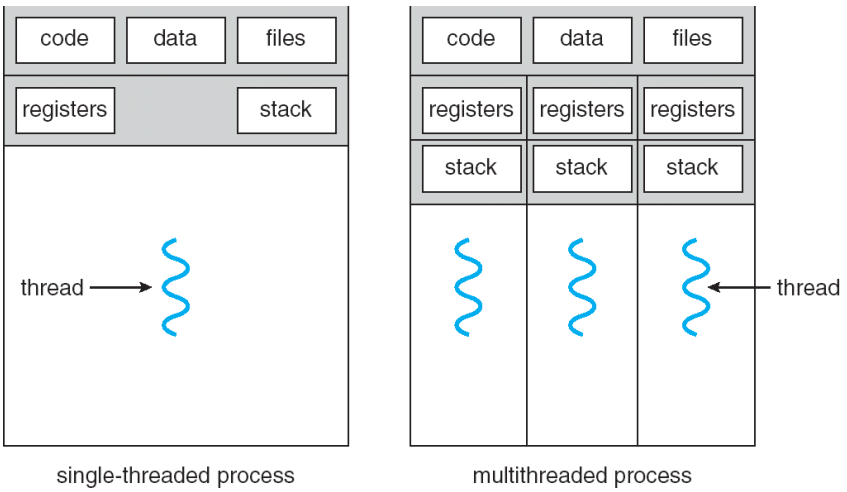
Threads differ from traditional multitasking operating system processes in that:

- processes are typically independent, while threads exist as subsets of a process
- processes carry considerably more state information than threads, whereas multiple threads within a process share process state as well as memory and other resources
- processes have separate address spaces, whereas threads share their address space
- processes interact only through system-provided inter-process communication mechanisms
- context switching between threads in the same process is typically faster than context switching between processes.

Cada thread tiene sus propios program counters, registros donde almacena sus variables,su stack y su estado. Pero comparten code, data y files.

Different threads in a process are not as independent as different processes. All threads have exactly the same address space, which means that they also share the same global variables. Since every thread can access every memory address within the process' address space, one thread can read, write, or even wipe out anothre thread's stack. There is no protection between threads because should not be necessary. All threads are owned by a single user, who has presumably created multiple threads so that the can cooperate, not fight. In addition to sharing an address space, all the threads can share the same set of open files, child processes, alarms, and signals.





Thread Control Block (TCB)
Hay informacion que pasa del PCB al (o los) TCB

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Like traditional process, thread can be in any of several states: running, blocked, ready or terminated. A running thread concurrently has the CPU and is active. A blocked thread is waiting for some event (or other thread) to unblock it. A ready thread is scheduled to run and will as soon as its turn come up. The transitions between threads states are the same as transitions between process states.

4.1. Aplicaciones de multithreading

Varias aplicaciones que concurren sobre los mismo datos como:

- Un server que lanza un thread por cada pedido.
- Un procesador de texto concurrente con su corrector y su armador de pagina.
- El manejo de Interfaces Gráficas.

Algunos de los sistemas hechos con threads podrían hacerse con eventos.
Eventos usan handlers y callbacks, los threads quedan congelados hasta que se les devuelve el control.
Escalar con threads es casi trivial, escalar con eventos tiene un techo (stack ripping).
No mezclar threads con eventos.

4.2. Implementacion de threading

There are two main ways to implement a threads package: in user space and in the kernel (a hybrid implementation is also possible).

4.2.1. Threads in user space

The kernel knows nothing about them. As far as the kernel is concerned, it is managing ordinary, single-threaded processes (can be implemented in operating systems than not suport threading). Threads are implemented by a library.

Cada proceso tiene su propia tabla de threads privada, para seguirle el rastro a cada uno de sus threads. En la tabla se guardan las propiedades del thread (program counter, stack pointer, registers, state, etc). La tabla es manejada por el runtime system.

When a thread does something that may cause it to become blocked locally, for example, waiting for another thread in its process to complete some work, it calls a run-time system procedure. This procedure checks to see if the thread must be put in a blocked state. If so, it stores the thread's registers in the thread table, looks in the table for a ready thread to run, and reloads the machine registers with the new thread's saved values. As soon as the stack pointer and program counter have been switched, the new thread comes to life again automatically.

La biblioteca de threads permite al usuario multiplexar su time slice.

Time slice: tiempo que puede ejecutarse un proceso sin que el SO lo pare (impide que el usuario bloquee el sistema).

Ventajas

- Doing thread switching like this is at least an order of magnitude (maybe more) faster than trapping to the kernel. The procedure that saves the thread's state and the scheduler are just local procedures, so invoking them is much more efficient than making a kernel call. Among other issues, no trap is needed, no context switch is needed, the memory cache need not be flushed, and so on. This makes thread scheduling very fast.

- Customized scheduling algorithm.
- Scale better, since kernel threads require some table space and stack space in the kernel, which can be a problem if there are a very large number of threads.

Problemas

- How to implement blocking systems calls. It should prevent one blocked thread from affecting others.
- If a thread starts running, no other thread in that process will ever run unless the first thread voluntarily gives up the CPU

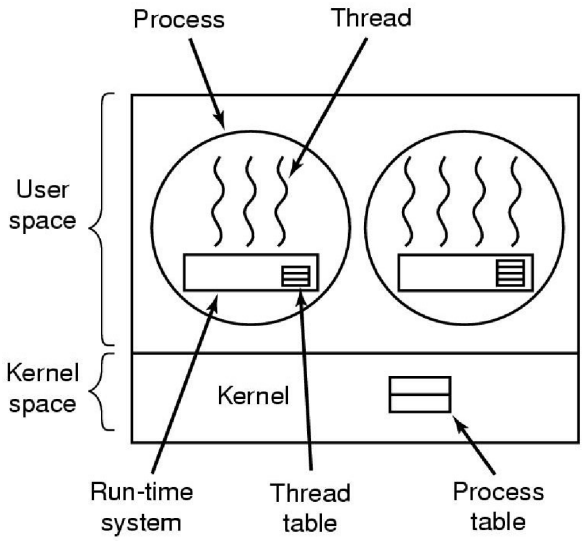


Figura 4: Threads en el espacio de usuario

4.2.2. Kernel threads

The kernel has a thread table that keeps track of all the threads in the system. When a thread wants to create a new thread or destroy an exisiting one, it makes a kernel call, which then does the creation or destruction by updating the kernel thread table.

La tabla del kernel es igual a la que se usa en los threads en el espacio de usuario, pero almacenada en el kernel.

All calls that might block a thread are implemented as system calls, at considerably greater cost than a call to a run-time system procedure. When a thread blocks, the kernel, at its options, can run either another thread from the same process (if one is ready) or a thread from a different process. With user-level threads, the run-time system keeps running threads from its own process until the kernel takes the CPU away from it (or there are no ready threads left to run).

Due to the relatively greater cost of creating and destroying threads in the kernel, some systems recycle their threads. When a thread is destroyed, it is marked as not runnable, but its kernel data structures are not otherwise affected. Later, when a new thread must be ccreated, and old thread is reactivated, saving some overhead.

Kernel threads do not require any new, nonblocking system calls.

Their main disadvantage is that the cost of a system call is substantial, so if thread operations (creation, termination, etc) are common, much more overhead will be incurred.

Some problems

- Multithreaded process fork: How many threads for the new one?
- Signals: when a signal comes in, which thread should handle it?

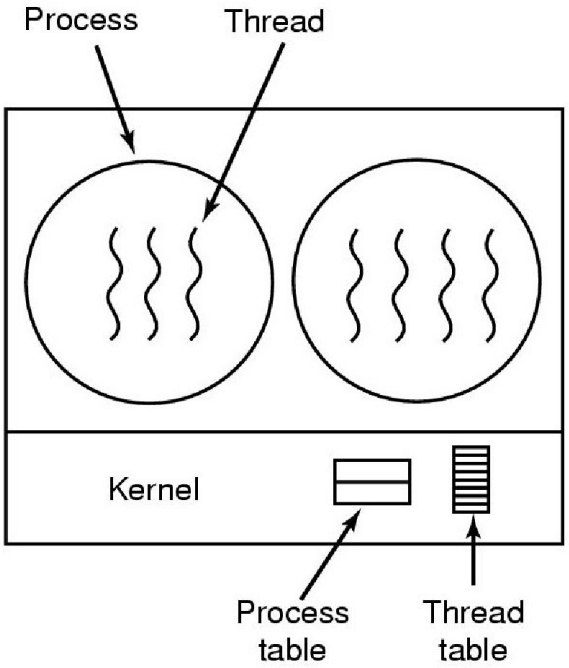


Figura 5: Kernel Threads

4.2.3. Implementaciones hibridas

One way is use kernel-level threads and then multiplex user-level threads onto some or all the kernel threads. The programmer can determine how many kernel threads to use and how many user-level threads to multiplex on each one.

The kernel is aware of only the kernel-level threads and schedules those. Some of those threads may have multiple user-level threads multiplexed on top of them. These user level threads are created, destroyed and scheduled just like user-level threads in a process.Each kernel-level thread has some set of user-level threads that take turns using it.

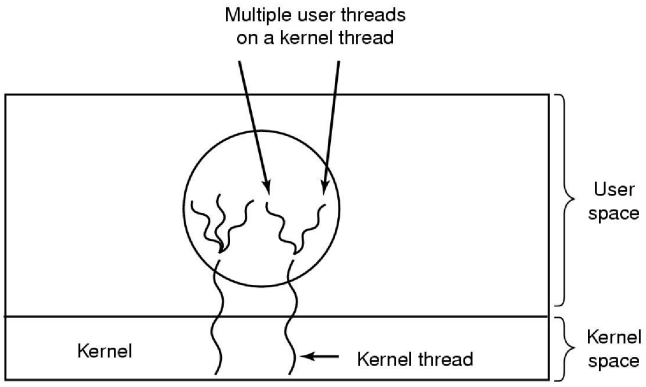


Figura 6: Threads híbridos

Scheduler activations

Mimic the funcionality of kernel threads but with better performance (similar to user space threads). Cuando se produce una interrupcion, el SO le devuelve el control al usuario.

The kernel assigns a certain number of virtual processors to each process and lets the user space runtime system allocate threads to processors.

The basic idea is that when the kernel knows that a thread has blocked, the kernel notifies the process runtime system by an “upcall”. The runtime system, at its own discretion, can either restart the blocked thread immediately or put it on the ready list to be run later.

An objection to scheduler activations is the fundamental reliance on upcalls, a concept tha violates the structure inherent in any layered system. Normally, layer n offers certain services that layer n+1 can call on, but layer n may not call procedures in layer n + 1. Upcalls do not follow this fundamental principle.

5. Administracion de memoria

5.1. Definición

Suministrar la memoria necesaria para el código de un programa y sus estructuras de datos. Esto incluye:

- Asignación de memoria.
- Reciclado del almacenamiento.

Está relacionado con el esquema de manejo de memoria soportado por el lenguaje, el compilador se encarga de intercalar el código correspondiente.

5.2. Vocabulario de manejo de memoria en los lenguajes

Declarar una variable Establecer nomenclatura (introducir el identificador) sin asignar memoria. Es un paso necesario pero no suficiente para usarla.

Definir una variable Asignarle memoria y posiblemente un valor inicial. Se puede usar una vez que está definida.

Ambiente Porción de código durante el cual una variable está declarada.

Vida Intervalo de la ejecución en el cual una variable tiene memoria asignada.

Ámbito (scope) Cuando una variable está en su ambiente y en su tiempo de vida. Es en tiempo de ejecucion.

5.3. Tipos de variables según su manejo en memoria

5.3.1. Externa

Se encuentra definida pero no declarada en el bloque. Debido a que las variables externas son accesibles globalmente, puede ser utilizadas para comunicar datos entre funciones, en lugar de una lista de parámetros. Además, como las variables externas permanecen en existencia, en vez de aparecer y desaparecer a lo largo de las llamadas y salidas de las funciones, retienen sus valores incluso después que las funciones que las funciones que le asignaron valor salieron.

5.3.2. Estática

Su vida se extiende a toda la duración del programa. When a program (executable or library) is loaded into memory, static variables are stored in the data segment of the program's address space (if initialized), or the BSS segment (if uninitialized), and are stored in corresponding sections of object files prior to loading. In terms of scope and extent, static variables have extent the entire run of the program, but may have more limited scope. A basic distinction is between a static global variable, which has global scope and thus is in context throughout the program, and a static local variable, which has local scope and thus is only in context within a function (or other local context).

5.3.3. Dinámica Automática

La memoria se asigna y libera automáticamente cuando la ejecución ingresa en el ámbito de la variable. Is a variable which is allocated and deallocated automatically when program flow enters and leaves the variable's context. No lo hace el programador, sino el compilador del lenguaje o una biblioteca.

5.3.4. Dinámica Controlada

La ejecución del programa (el programador) controla explícitamente cuando se asigna memoria a la variable.

Garbage Collection Cuando la recuperación de memoria no referenciada es automática.

Liberación manual Cuando la recuperación se hace en el código del programa

5.4. Organización del almacenamiento en tiempo de ejecución

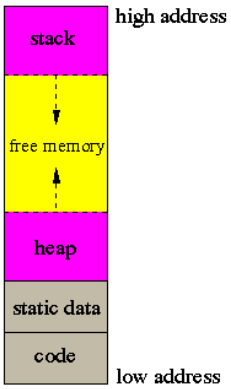
Las variables dinámicas automáticas (o lexically scoped) se manejan por medio de un *stack* (pila).

Because the data is added and removed in a last-in-first-out manner, stack-based memory allocation is very simple and typically faster than heap-based memory allocation (also known as dynamic memory allocation). Another feature is that memory on the stack is automatically, and very efficiently, reclaimed when the function exits, which can be convenient for the programmer if the data is no longer required. If however, the data needs to be kept in some form, then it must be copied from the stack before the function exits. Therefore, stack based allocation is suitable for temporary data or data which is no longer required after the creating function exits.

Mantiene las variables de llamar de un procedimiento a otro.

Las funciones recursivas hacen que crezca mucho el stack.

Las variables dinámicas controladas se manejan por medio de una estructura llamada heap. Que se puede implementar de varias formas (Listas o Buddies)



This is the layout in memory of an executable program. Note that in a virtual memory architecture (which is the case for any modern operating system), some parts of the memory layout may in fact be located on disk blocks and they are retrieved in memory by demand (lazily).

The machine code of the program is typically located at the lowest part of the layout. Then, after the code, there is a section to keep all the fixed size static data in the program. The dynamically allocated data (ie. the data created using malloc in C) as well as the static data without a fixed size (such as arrays of variable size) are created and kept in the heap. The heap grows from low to high addresses. When you call malloc in C to create a dynamically allocated structure, the program tries to find an empty place in the heap with sufficient space to insert the new data; if it can't do that, it puts the data at the end of the heap and increases the heap size.

The focus of this section is the stack in the memory layout. It is called the run-time stack. The stack, in contrast to the heap, grows in the opposite direction (upside-down): from high to low addresses, which is a bit counterintuitive. The stack is not only used to push the return address when a function is called, but it is also used for allocating some of the local variables of a function during the function call, as well as for some bookkeeping.

Let's consider the lifetime of a function call. When you call a function you not only want to access its parameters, but you may also want to access the variables local to the function. Worse, in a nested scoped system where nested function definitions are allowed, you may want to access the local variables of an enclosing function. In addition, when a function calls another function, we must forget about the variables of the caller function and work with the variables of the callee function and when we return from the callee, we want to switch back to the caller variables. That is, function calls behave in a stack-like manner.

El Stack mantiene el Registro de Activación o frame. A call stack is composed of stack frames (also called activation records or activation frames). These are machine dependent and ABI-dependent data structures containing subroutine state information. Each stack frame corresponds to a call to a subroutine which has not yet terminated with a return. El *stack frame* usualmente incluye al menos los siguientes items (en orden en que se encolan):

- Argumentos (parámetros) pasados a la rutina (si posee alguno)
- Dirección de retorno a la rutina invocadora (ej. En el stack frame de una función `DibujarLinea`, una dirección al código de la función que la llamó, como puede ser `DibujarCuadrado`)

- Espacio para las variables locales de la rutina (si existen).

Para el procesador es dificil manejar la memoria: no toda la memoria es igual.

El uso de la memoria cache no depende del programador ni del sistema operativo. Sólo la ve el hardware (la microarquitectura).

A CPU cache is a cache used by the central processing unit (CPU) of a computer to reduce the average time to access data from the main memory. The cache is a smaller, faster memory which stores copies of the data from frequently used main memory locations. Most CPUs have different independent caches, including instruction and data caches, where the data cache is usually organized as a hierarchy of more cache levels (L1, L2 etc.)

5.4.1. Ciclo de instruccion

An instruction cycle (sometimes called fetch-and-execute cycle, fetch-decode-execute cycle, or FDX) is the basic operation cycle of a computer. It is the process by which a computer retrieves a program instruction from its memory, determines what actions the instruction requires, and carries out those actions. This cycle is repeated continuously by the central processing unit (CPU), from bootup to when the computer is shut down.

5.5. Modos de direccionamiento

5.5.1. Modos de direccionamiento para código

El compilador genera las direcciones de las instrucciones:

- Puede ser una dirección absoluta ⇒ secuencial.
- Puede ser relativa al Program Counter (position independent) ⇒ secuencial
- Puede generar las direcciones en cada instrucción como SECD para cálculo

5.5.2. Modos de direccionamiento para datos

- Dirección Absoluta
- Operador Inmediato (literal)
- Dirección indirecta (*ptr)
- Dirección relativa a Program Counter (*+ptr)
- Base/Índice/Offset.
- Base + Desplazamiento

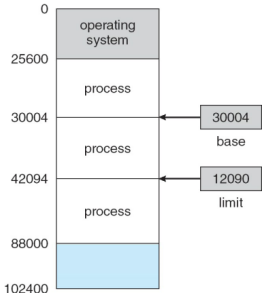
5.6. Protección de memoria

El Sistema Operativo debe impedir a un proceso invadir la memoria de otro.

Un espacio de direcciones es un set de direcciones que un proceso puede usar para direccionar memoria. Cada proceso tiene su propio set de direcciones, independiente de aquellos que pertenecen a otro proceso.

Una forma de hacerlo es por medio de un *registro base* y uno *límite*. El registro base puede usarse para direccionamiento indirecto. Cada dirección se compara con ambos límites.

El Sistema Operativo no tiene restricciones al operar en Modo Kernel



5.7. Administracion de memoria por el sistema operativo

El Modelo de Procesos añade un área especial para la administración del proceso (La U_Area).
El Sistema Operativo debe proveer alojamiento para la ejecución del proceso (Memory Allocation).
En multiprocesamiento, más de un proceso hace sus requerimientos de memoria.

5.7.1. Segmentación

- Es una forma de proveer de más de un espacio de direcciones.
- El programador (o el compilador) es quien lo usa.
- Su uso puede superponerse con el del paginado.
- La protección puede asociarse a segmentos.
- Algunos sistemas separan las tablas de segmentos (ej. Intel):
  - Global (para el Sistema Operativo)
  - Local (para los procesos). O hasta una por proceso.

Esto permite que los procesos no compitan con el Sistema Operativo por memoria.

5.8. Administración del espacio libre/ocupado

Memory allocation is the process of assigning blocks of memory on request. Typically the allocator receives memory from the operating system in a small number of large blocks that it must divide up to satisfy the requests for smaller blocks. It must also make any returned blocks available for reuse. There are many common ways to perform this, with different strengths and weaknesses.

Pueden usarse bitmaps o listas encadenadas. Aparecen distintos Algoritmos de alojamiento: *Best Fit*, *Worst Fit*, *First Fit*, *BuddySystem*, *Swapping*.

5.8.1. Best fit

Buscar el hueco mas ajustado.

The allocator keeps a list of free blocks (known as the free list) and, on receiving a request for memory, scans along the list for the first block that is large enough to satisfy the request. If the chosen block is significantly larger than that requested, then it is usually split, and the remainder added to the list as another free block. The first fit algorithm performs reasonably well, as it ensures that allocations are quick.

5.8.2. Worst fit

Buscar el hueco más holgado.

This approach encourages external fragmentation, but allocation is very fast.

5.8.3. First fit

Buscar el primer hueco en que quepa.

The free block with the "tightest fit" is always chosen. The fit is usually sufficiently tight that the remainder of the block is unusably small.

Tienen rendimientos parecidos, pero generalmente First Fit funciona un poco mejor.

5.8.4. Buddy system

Usado en los dispositivos modernos que no tienen memoria virtual (la memoria fisica es la que ven los procesos). La memoria se asigna en cantidades potencias de dos.

Permite una recuperación rápida de huecos grandes. Sufre de fragmentación interna. Su implementación es muy sencilla y rápida.

No se usa en SO de uso general.

In a buddy system, the allocator will only allocate blocks of certain sizes, and has many free lists, one for each permitted size. The permitted sizes are usually either powers of two, or form a Fibonacci sequence (see

below for example), such that any block except the smallest can be divided into two smaller blocks of permitted sizes. When the allocator receives a request for memory, it rounds the requested size up to a permitted size, and returns the first block from that size's free list. If the free list for that size is empty, the allocator splits a block from a larger size and returns one of the pieces, adding the other to the appropriate free list.

When blocks are recycled, there may be some attempt to merge adjacent blocks into ones of a larger permitted size (coalescence). To make this easier, the free lists may be stored in order of address. The main advantage of the buddy system is that coalescence is cheap because the "buddy" of any free block can be calculated from its address.

5.8.5. Swapping

Consiste en pasar de memoria principal (RAM) a memoria secundaria (flash o disco rígido) un proceso que no está corriendo y que hace mucho que no está activado (ej: esperando intervencion humana). Se baja todo el proceso. Pueden pasarse procesos bloqueados o disponibles pero sin ejecutarse.

Debe tenerse en cuenta la interacción con la I/O.

Se desplaza todo el proceso y se marca en el PCB (process control block). Lo decide el SO.

Se hace cuando no hay más memoria principal disponible.

Al hacer swap-in, un proceso puede volver a una dirección distinta. El direccionamiento indirecto resuelve la reubicación.

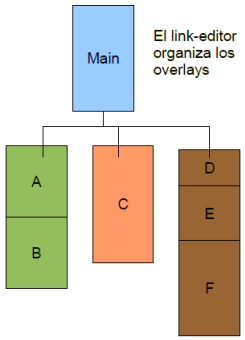
6. Memoria virtual

6.1. Overlays

Solución para correr programas demasiado grandes para la memoria disponible. Se comparten partes de memoria. Reemplaza un bloque de código por otro.

El programador debe planificar el uso de *overlays* según el uso de las rutinas del sistema.

Aún se usa en PDAs, Celulares y Embedded.



Main puede llamar a cualquier procedimiento. **A** puede llamar a **B**, pero no a **C** ni a **D**, **E** o **F**. **C** no puede llamar a ninguno. Cada una de las tres partes son excluyentes

6.2. Memoria virtual

Evita que el programador deba planificar el uso de *overlays*.

Basicamente, utilizando *memoria virtual*, cada programa tiene su propio espacio de direcciones, que está fragmentado en bloques llamados *páginas*. Cada página es un rango continuo de direcciones. Estas páginas están mapeadas en memoria física, pero no todas las páginas tienen que estar en memoria (física) para correr el programa.

Cuando el programa referencia una parte de su espacio de direcciones que está presente en memoria física, el hardware ejecuta el mapeo necesario al instante. En el caso que la parte no esté en memoria física, el sistema operativo es alertado para que busque la pieza faltante y reejecute la instrucción fallida.

Las principales ventajas de la memoria principal incluyen:

- Liberar a las aplicaciones de tener que administrar un espacio de memoria compartido.
- Mejor seguridad debido al aislamiento de la memoria
- Poder ser capaz de, conceptualmente, utilizar más memoria que la disponible físicamente, utilizando la técnica de *paginación*

Virtual memory makes application programming easier by hiding fragmentation of physical memory; by delegating to the kernel the burden of managing the memory hierarchy (eliminating the need for the program to handle overlays explicitly); and, when each process is run in its own dedicated address space, by obviating the need to relocate program code or to access memory with relative addressing. Most virtual memory systems use a technique called paging.

6.3. Páginas

A page, memory page, or virtual page is a fixed-length contiguous block of virtual memory, described by a single entry in the page table. It is the smallest unit of data for memory allocation performed by the operating system on behalf of a program, and for transfers between the main memory and any other auxiliary store, such as a hard disk drive.

Virtual memory allows a page that does not currently reside in main memory to be addressed and used. If a program tries to access a location in such a page, an exception called a page fault is generated. The hardware or operating system is notified and loads the required page from the auxiliary store (hard disk) automatically. A program addressing the memory has no knowledge of a page fault or a process following it. Thus a program can

address more (virtual) RAM than physically exists in the computer. Virtual memory is a scheme that gives users the illusion of working with a large block of contiguous memory space (perhaps even larger than real memory), when in actuality most of their work is on auxiliary storage (disk). Fixed-size blocks (pages) or variable-size blocks of the job are read into main memory as needed.

A transfer of pages between main memory and an auxiliary store, such as a hard disk drive, is referred to as paging.

Las direcciones generadas por la CPU se dividen bloques de tamaño fijo llamados páginas.

Estas direcciones se llaman direcciones virtuales.

Generalmente de 2 o 4 KB para evitar fragmentación interna.

Esta técnica permite el uso de memoria no contigua.

Las páginas se guardan en disco en un *page data set*.

6.4. Marcos de páginas (frames)

La memoria principal se divide en frames, del mismo tamaño que las páginas.

Las direcciones de los frames se llaman direcciones reales.

Una unidad de Hardware, llamada *MMU* (Memory Management Unit) mapea las direcciones virtuales en reales.

6.4.1. Ejemplo de paging

Generalmente las paginas y los frames son del mismo tamaño. Por ejemplo 4KB.

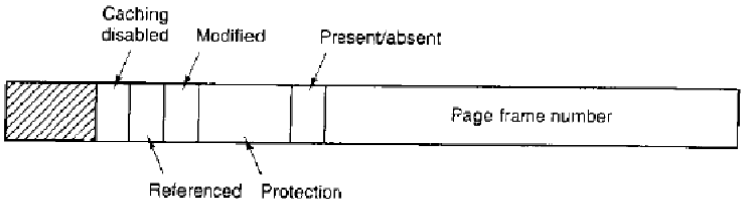
Si tenemos 64 KB de memoria virtual y 32 KB de memoria fisica, tenemos 16 páginas virtuales y 8 frames. De esta forma solo 8 páginas virtuales pueden estar mapeadas. Si el programa hace referencia a una pagina que no esta mapeada, el CPU hace un trap (*page fault*) al SO, para reemplazar una pagina mapeada (poco usada) por la pagina nueva. Se escribe la pagina vieja en disco y se mapea la nueva.

6.5. Page table

Mapea las paginas con los frames.

6.5.1. Estructura

La estructura de una entrada en la tabla de paginación puede variar de maquina a maquina. Ejemplo:



Page frame number

Present / absent bit Si es 1, la entrada es válida y se puede usar. Si es 0, la pagina virtual no está actualmente en memoria. Acceder a una página cuyo bit está en 0 causa un page fault.

Protection bit Dice que tipo de acceso tiene permitido (lectura, lectura/escritura).

Modified ("dirty") bit cuando se escribe en una pagina, el hardware automaticamente setea el bit Modified en 1. Este bit se lee cuando el sistema opeartivo decide bajar esta página de memoria. Si la pagina fue modifica (es "dirty"), se debe guardar de vuelta en disco. Si no fue modificada ("clean"), se puede directamante abandonar, ya que la copia que está en disco es válida.

Referenced bit Se setea cuando una pagina es referenciada, ya sea para lectura o escritura. Ayuda al sistema operativo a elegir que pagina dar de baja. Las páginas que no están siendo usadas son mejores candidatas a ser dadas de baja.

Caching Se puede habilitar o deshabilitar el cacheo de la pagina.

6.5.2. Tabla de páginas directa

Se utiliza una *tabla hash*, donde el valor del hash es el número de la página virtual. Cada entrada en la tabla hash contiene una lista enlazada de elemento que dirigen a la misma posición.

6.5.3. Tabla de página multinivel

Cuando crece mucho el espacio de memoria se usan tablas que referencias otras tablas y estas a su vez referencian a los frames como en el caso de paginación directo.

Se tiene en memoria las tablas que están siendo utilizadas y se deja en disco aquellas que no están siendo referenciadas.

6.5.4. Tabla de página invertidas

Es otra solución al problema del crecimiento del espacio de memoria.

En este diseño en vez de tener una tabla con cada espacio de memoria virtual referenciando su frame en memoria real es al revés.

La clave de la tabla es el frame, es decir que hay tantas entradas en la tabla como espacio de memoria real.

Se ahorra mucho espacio pero el cálculo de dirección es más costoso, para encontrar una dirección hay que iterar sobre la tabla buscando la dirección.

En la práctica el TBL guarda las páginas más utilizadas.

6.6. Memory management unit (MMU)

Es el responsable de traducir las direcciones virtuales (o lógicas) a direcciones reales (o físicas). En general hace uso de una cache asociativo, el *Translation Lookaside Buffer (TLB)*.

This solution is based on the observation that most programs tend to make a large number of references to a small number of pages, and not the other way around. Thus only a small fraction of the page table entries are heavily read; the rest are barely used at all.

The solution is a small hardware device for mapping virtual addresses to physical addresses without going through the page table. It is usually inside the MMU and consits of a small number of entries. Each entry contains information about one page, including the virtual page numbet, a bit that is set when the page is modified, the protection code (read/write/execute permissions), and the physical page frame in which the page is located. These fields have a one-to-one correspondence with the fields in the page table, except for the virtual page number, which is not needed in the page table. Another bit indicates wheter the entry is valid (i.e, in use) or not.

When a virtual address is presented to the MMU for translation, the hardware first checks to see if its virtual page number is present in the TLB by comparing it to all the entries simultaneously (in parallel). If a valid math is found, the page frame is taken directly from the TLB, without going to the page table.

If the virtual page number is not in the TLB, the MMU detects the miss and does an ordinary page table lookup. It then evicts one of the entries from the TLB and replaces it with the page table entry just looked up. Thus if that page is used again soon, the second time it will result in a TLB hit rather than a miss. When an entry is purged from the TLB, the modified bit is copied back into the page table entry in memory. The other values are already there, except the reference bit. When the TLB is loaded from the page table, all the fields are taken form memory. Si no está el hardware en el *MMU*, existen algunas soluciones para implementar el TLB con software.

Hit/miss ratio relacion entre aciertos y fracasos.

Oportunidades de optimizacion

- Algoritmos de paginado.
- Que partes pasar al Hard.
- Como evitar los page faults.
- Estructura de las tablas de páginas.
- Organización del Page Data Set.
- Algunas otras cosas que se resuelven con el paginado.

6.7. Algoritmos de reemplazo de páginas

1. The theoretically optimal page replacement algorithm
2. Not recently used
3. FIFO
4. Second-chance
5. Clock
6. Least recently used
7. Random
8. Not frequently used
9. Aging

6.7.1. The theoretically optimal page replacement algorithm

The theoretically optimal page replacement algorithm is an algorithm that works as follows: when a page needs to be swapped in, the operating system swaps out the page whose next use will occur farthest in the future.

This algorithm cannot be implemented in the general purpose operating system because it is impossible to compute reliably how long it will be before a page is going to be used, except when all software that will run on a system is either known beforehand and is amenable to the static analysis of its memory reference patterns, or only a class of applications allowing run-time analysis.

6.7.2. Not recently used

The not recently used (NRU) page replacement algorithm is an algorithm that favours keeping pages in memory that have been recently used. This algorithm works on the following principle: when a page is referenced, a referenced bit is set for that page, marking it as referenced. Similarly, when a page is modified (written to), a modified bit is set. The setting of the bits is usually done by the hardware, although it is possible to do so on the software level as well.

At a certain fixed time interval, the clock interrupt triggers and clears the referenced bit of all the pages, so only pages referenced within the current clock interval are marked with a referenced bit. When a page needs to be replaced, the operating system divides the pages into four classes:

3 referenced, modified

2 referenced, not modified

1 not referenced, modified

0 not referenced, not modified

Although it does not seem possible for a page to be not referenced yet modified, this happens when a class 3 page has its referenced bit cleared by the clock interrupt. The NRU algorithm picks a random page from the lowest category for removal. So out of the above four pages, the NRU algorithm will replace the not referenced, not modified. Note that this algorithm implies that a modified but not referenced (within last clock interval) page is less important than a not modified page that is intensely referenced.

6.7.3. FIFO

Requiere mantener una cola de páginas.

No es buena idea.

Reemplaza las páginas del scheduler o del Kernel.

Este algoritmo experimenta la anomalía de Belady (the phenomenon where increasing the number of page frames results in an increase in the number of page faults for a given memory access pattern)

6.7.4. Second-chance

A modified form of the FIFO page replacement algorithm, known as the Second-chance page replacement algorithm, fares relatively better than FIFO at little cost for the improvement. It works by looking at the front of the queue as FIFO does, but instead of immediately paging out that page, it checks to see if its referenced bit is set. If it is not set, the page is swapped out. Otherwise, the referenced bit is cleared, the page is inserted at the back of the queue (as if it were a new page) and this process is repeated. This can also be thought of as a circular queue. If all the pages have their referenced bit set, on the second encounter of the first page in the list, that page will be swapped out, as it now has its referenced bit cleared. If all the pages have their reference bit set then second chance algorithm degenerates into pure FIFO.

As its name suggests, Second-chance gives every page a "second-chance" – an old page that has been referenced is probably in use, and should not be swapped out over a new page that has not been referenced.

6.7.5. Clock

Clock is a more efficient version of FIFO than Second-chance because pages don't have to be constantly pushed to the back of the list, but it performs the same general function as Second-Chance. The clock algorithm keeps a circular list of pages in memory, with the "hand" (iterator) pointing to the last examined page frame in the list. When a page fault occurs and no empty frames exist, then the R (referenced) bit is inspected at the hand's location. If R is 0, the new page is put in place of the page the "hand" points to, otherwise the R bit is cleared. Then, the clock hand is incremented and the process is repeated until a page is replaced.

6.7.6. Least recently used

Cada referencia a memoria actualiza un timestamp en la page table. Se reemplaza la página cuyo time stamp sea el más antiguo. Requiere bastante auxilio de hard para tener una performance adecuada.

6.7.7. Random

Random replacement algorithm replaces a random page in memory. This eliminates the overhead cost of tracking page references. Usually it fares better than FIFO, and for looping memory references it is better than LRU, although generally LRU performs better in practice.

6.7.8. Not frequently used

The not frequently used (NFU) page replacement algorithm requires a counter, and every page has one counter of its own which is initially set to 0. At each clock interval, all pages that have been referenced within that interval will have their counter incremented by 1. In effect, the counters keep track of how frequently a page has been used. Thus, the page with the lowest counter can be swapped out when necessary.

Problema: No tiene en cuenta el tiempo. Nunca olvida!

6.7.9. Aging

The aging algorithm is a descendant of the NFU algorithm, with modifications to make it aware of the time span of use. También tiene un contador, por cada referencia se enciende el primer bit. Cuando se produce una interrupción de reloj (0,02 s), el SO desplaza todos los bits una posición a la derecha. Cuando ocurre un fallo se cambia la página de menor valor en el contador.

6.8. Otros conceptos

6.8.1. Working set

Los programas exhiben un comportamiento conocido como localidad de referencia. En cada fase de su ejecución, el proceso referencia solo a un pequeño número de páginas (no necesariamente contiguas). Ese conjunto se llama Working Set (aunque la definición cambia con la implementación). El Working Set va cambiando a medida que progresa la ejecución.

6.8.2. Pre-paginado

Si un proceso pagina durante mas tiempo que el que ejecuta se dice que hace thrashing. This leads to low CPU utilization.

Posible solucion: Se pre-pagina (cargan en memoria) todas las páginas del último Working Set del proceso. Se re-calcula el Working Set a intervalos.

6.8.3. Tablas de páginas Locales y Globales

Algunos Sistemas permiten el reemplazo de cualquier página (paginado global).

Otros solo permiten que un proceso pagine sobre si mismo, evitando efectos en la performance del resto. Esta estrategia es la que se usaría en un Sistema Operativo orientado a Objetos.

En general, el paginado global funciona mejor. Ya que no todos los procesos tienen los mismos requerimientos.

6.8.4. Archivos de paginado

Una página no es una buena unidad de transferencia.

Se usan entonces láminas (slab) de páginas en cada transferencia. La ubicación de una página es entonces #slab + offset. Las slabs se acomodan en el page data set (o partición de paginado). Que se formatea y ubica por anticipado.

6.8.5. Paginado de código

Las páginas de código son read-only.

Se paginan directamente desde el archivo del programa ejecutable.

El elf tiene previsiones para ello.

Es una forma muy eficiente de cargar un programa a memoria. El resto de las páginas tiene su “shadow” en disco.

6.8.6. Otros tópicos

El paginado interactúa con la I/O. Se deben fijar las páginas donde hay transferencia desde memoria secundaria. Los archivos pueden accederse como memoria virtual.

7. Linkers y loaders

7.1. Linker

7.1.1. Definición

Un *linker* o *link editor* es un programa que toma uno o más *archivos objeto* generados por un compilador y los combina en un único archivo ejecutable, o en un archivo objeto más.

Un archivo objeto es un archivo que contiene *código objeto*, es decir, código de máquina reacomodable que usualmente no se puede ejecutar directamente. Los archivo objeto son producidos por un *ensamblador*, *compilador* u otro traductor de lenguajes.

Programa fuente (1 o varios) ⇒ Traductor ⇒ Programa objeto (1 o varios)

7.1.2. Traducción - ensamblado

- Si el lenguaje es un assembler, la traducción es un ensamblado (assembly) hecho por un programa ensamblador (assembler).
- Convierte código de lenguaje ensamblador memotécnico a códigos de operación (machine language instruction).
- Resuelve identificadores a posiciones de memoria.
- Algunos proveen abstracciones de programación avanzadas. Existe un assembler distinto para cada arquitectura, incluso hay assemblers generales.

7.1.3. Traducción - compilación

Si se trata de un lenguaje de alto o mediano nivel, la traducción es una compilación.

Lenguaje fuente ⇒ Traductor (compilador) ⇒ Lenguaje objeto (.o) (microarquitectura)

7.1.4. Link-editor

Entradas

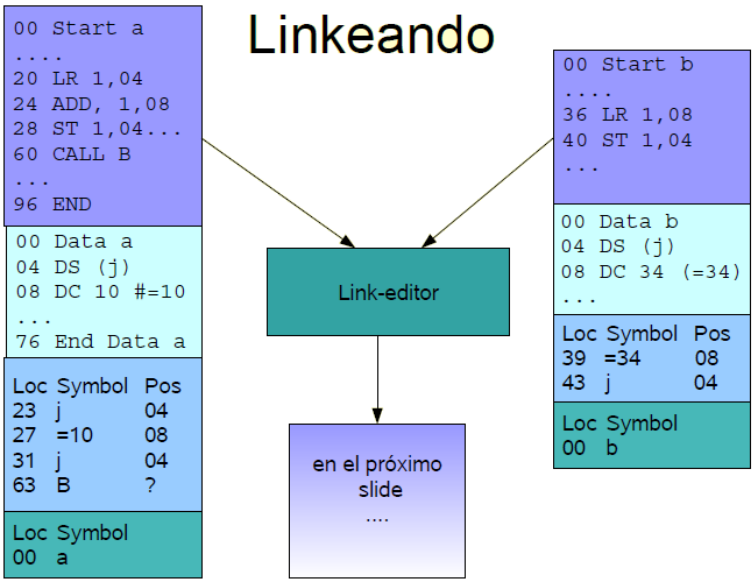
- Programas objeto
- Bibliotecas
- Programas ejecutables: para usarlas como apis. No recomendable.

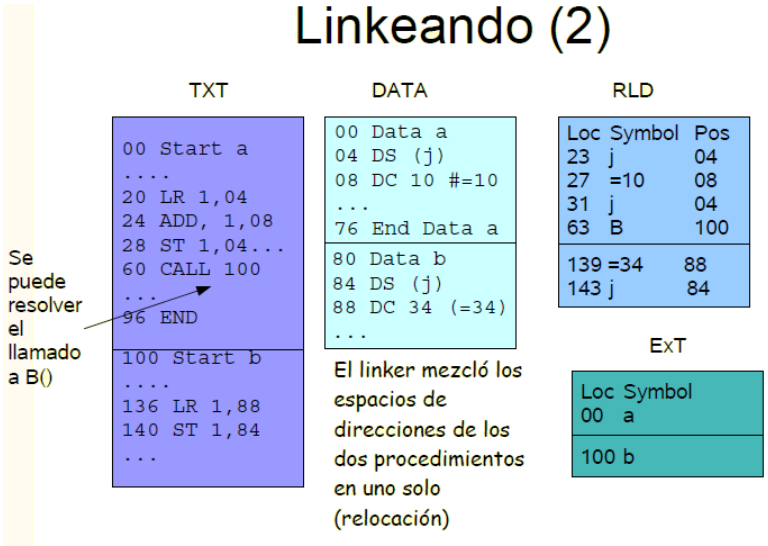
Salidas

- Programas ejecutables
- Programas objeto
- Bibliotecas

Mezcla las direcciones de cada módulo en un único espacio de direcciones.

Biblioteca y programa objeto no ejecutable son diferentes conceptualmente (de objetivos de creación): La biblioteca es pensada para usarse en diferentes programas, mientras que el programa objeto se crea para un programa en particular.





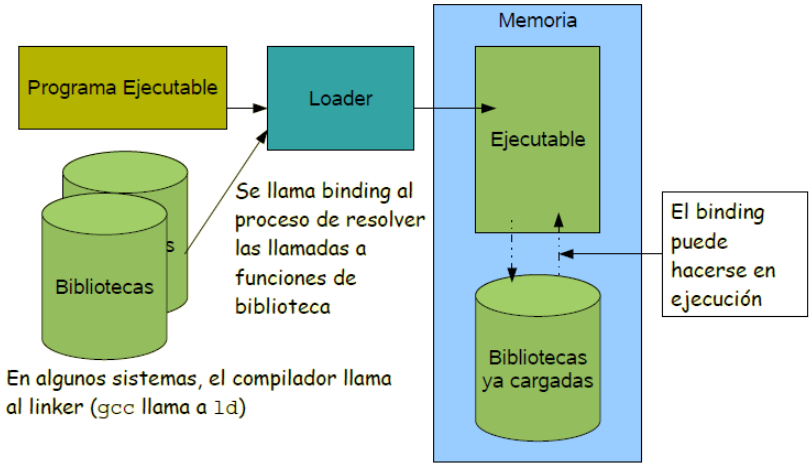
7.2. Loader

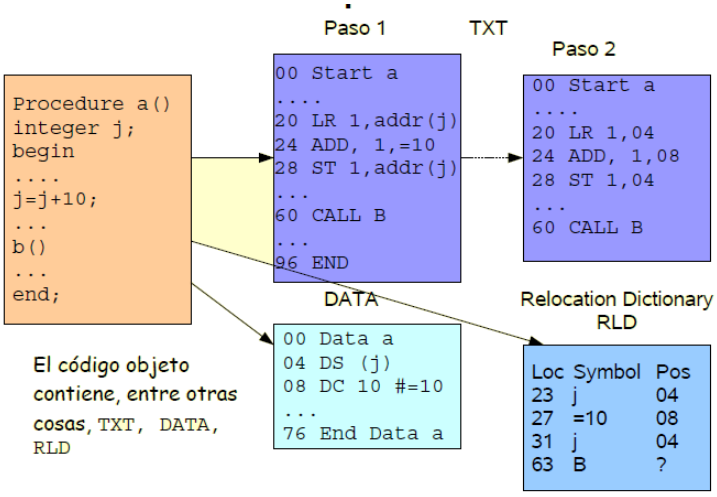
En el linker, el ejecutable puede incluir las bibliotecas o no

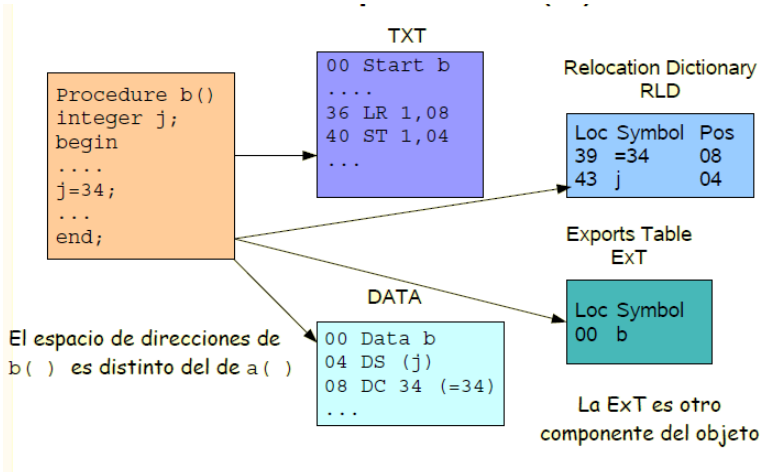
- Si lo incluye (forma *estática*), el programa ya incluye los símbolos y están cargados en memoria.
- Si no se incluyen, las llamadas a la biblioteca se resuelven en tiempo de ejecución (forma *dinámica*). El **loader** vincula las llamadas con la biblioteca.

Permite cambiar el funcionamiento actualizando las librerías del SO. Pero depende de que la biblioteca exista en el SO, y existen problemas de seguridad por ejecutar el código externo.

Linker y loader son en momento de post-compilación.







7.3. Object file formats (OFF)

Es clave para la performance del sistema.
Algunos prevén su interacción con el paginado.
Se suele utilizar el mismo formato para ejecutables, objetos y bibliotecas.

Que dos Sistemas Operativos tengan el mismo OFF no significa que los programas de uno puedan correr en el otro.

com

- Se usan en DOS.
- Son los que tienen extensión .com. Microsoft los llama bin o binary file
- Se cargan en una dirección fija de memoria (0x100).
- Datos y código están en el mismo segmento.
- Son monousuario y monoproceso

exe

- Aparece en DOS 2.0.
- Su primer byte o (Magic Number) es "MZ", iniciales de Mark Zbikowski.
- Tiene previsión para relocación en memoria.

Common Object File Format (COFF)

- Aparece en Unix pero se usa en otros ambientes.
- Se compone de varias secciones separadas por headers (con limitación de longitud).
- Se usa para bibliotecas (Aunque no de enlace dinámico).
- Soporta debug (pero solo de C): nm(1) lo puede inspeccionar.

Windows Portable Executable (PE)

- Es una adaptación del COFF. El Windows hace un wrapping del COFF.
- Su magic es "PE" pero comienza con un "MZ" por "compatibilidad".
- Tiene definido espacio para resources.
- Tiene definido tablas para el uso de bibliotecas compartidas.

Executable and Linkable Format (ELF)

Debido a su diseño, ELF es flexible y extensible, y no está atado a ningún procesador o arquitectura en particular. Esto le permitió ser adoptado por muchos sistemas operativos en distintas plataformas, convirtiéndolo en un estandar de facto.

Cada ELF posee un ELF header, seguido de los datos del archivo. Estos datos pueden incluir:

- Program header table, describing zero or more segments
- Section header table, describing zero or more sections
- Data referred to by entries in the program header table or section header table

The segments contain information that is necessary for runtime execution of the file, while sections contain important data for linking and relocation. Any byte in the entire file can be owned by at most one section, and there can be orphan bytes, which are not owned by any section.

El ELF header define si se utilizan direcciones de 32 o 64 bits.

Sirve para ejecutables y bibliotecas.

Un directorio permite agregar nuevas secciones.

Tiene previsiones para emulación.

Binarios Universales

Tiene los códigos de las distintas arquitecturas y cuando se ejecuta decide cual usa.

A fat binary (or multiarchitecture binary) is a computer executable program, which has been expanded (or "fattened") with code native to multiple instruction sets which can consequently be run on multiple processor types. The usual method of implementation is to include a version of the machine code for each instruction set, preceded by code compatible with all operating systems, which executes a jump to the appropriate section. This results in a file larger than a normal one-architecture binary file, thus the name.

7.4. Segmentación de memoria en OFF

Memory segmentation is the division of a computer's primary memory into segments or sections. In a computer system using segmentation, a reference to a memory location includes a value that identifies a segment and an offset within that segment. Segments or sections are also used in object files of compiled programs when they are linked together into a program image and when the image is loaded into memory.

Segments usually correspond to natural divisions of a program such as individual routines or data tables so segmentation is generally more visible to the programmer than paging alone. Different segments may be created for different program modules, or for different classes of memory usage such as code and data segments. Certain segments may be shared between programs.

8. Bibliotecas (Libraries)

Las bibliotecas son una *colección de subprogramas* usados en el desarrollo de Software.
Contienen "código auxiliar" y datos que brindan servicios a distintos programas.
Permiten el desarrollo modular.
Convierten los servicios en "commodities" entre los cuales se puede elegir.
The value of a library is the reuse of the behavior. When a program invokes a library, it gains the behavior implemented inside that library without having to implement that behavior itself. Libraries encourage the sharing of code in a modular fashion, and ease the distribution of the code.

8.1. Tipos de bibliotecas

Se clasifican según el tiempo en que se cargan al programa (y por ende la ubicación del código fuente y quien lo hace)

1. Estáticas
2. Dinámicas
3. Dinámicas compartidas

8.1.1. Estáticas

Los subprogramas se incluyen en el ejecutable. El link-editor es quien las incorpora al archivo ejecutable. Los símbolos quedan definidos.

Ventajas

- Facilitan la instalación de los programas en otras máquinas, ya que la aplicación queda contenida en un único archivo ejecutable.
- The application can be certain that all its libraries are present and that they are the correct version. This avoids dependency problems, known colloquially as DLL Hell or more generally dependency hell.
- Evitan que los troyanos ataquen programas sensibles (se sabe a exactamente qué código se esta corriendo, cosa que no pasa con las bibliotecas dinámicas).
- It is enough to include those parts of the library that are directly and indirectly referenced by the target executable (or target library). With dynamic libraries, the entire library is loaded, as it is not known in advance which functions will be invoked by applications. Whether this advantage is significant in practice depends on the structure of the library.

Desventajas

- In static linking, the size of the executable becomes greater than in dynamic linking, as the library code is stored within the executable rather than in separate files. But if library files are counted as part of the application then the total size will be similar.

8.1.2. Dinámicas

El link-editor solo indica las llamadas. Éstas las resuelve el Loader durante la carga o durante la ejecución. Si es en ejecución, no carga la biblioteca cuando se ejecuta, sino cuando la necesita.

Ventajas

- Permite cambiar el funcionamiento actualizando las librerías del SO.
- Genera ejecutables más pequeños.

Desventajas

- Depende de que la biblioteca exista en el SO.

- Problemas de seguridad por ejecutar código externo.

Se debe solucionar el problema de la *relocación*: No se puede depender de posiciones "absolutas" de memoria ni en el programa ni en las bibliotecas. Se deben calcular las direcciones cada vez que se carga el programa o la biblioteca.

En general se usa una Import Table como acceso indirecto a las direcciones de la biblioteca para simplificar el proceso de linking.

Relocación

Relocation is the process of assigning load addresses to various parts of a program and adjusting the code and data in the program to reflect the assigned addresses. A linker usually performs relocation in conjunction with symbol resolution, the process of searching files and libraries to replace symbolic references or names of libraries with actual usable addresses in memory before running a program.

Relocation is typically done by the linker at link time, but it can also be done at run time by a relocating loader, or by the running program itself. Some architectures avoid relocation entirely by deferring address assignment to run time; this is known as zero address arithmetic.

Relocation is typically done in two steps:

1. Each object file has various sections like code, data, .bss etc. To combine all the objects to a single executable, the linker merges all sections of similar type into a single section of that type. The linker then assigns run time addresses to each section and each symbol. At this point, the code (functions) and data (global variables) will have unique run time addresses.

2. Each section refers to one or more symbols which should be modified so that they point to the correct run time addresses based on information stored in a relocation table in the object file.

8.1.3. Dinamicas Compartidas (DLL de windows)

- Están una sola vez en memoria. La comparten todos los procesos que la usan.
- Se guardan en un PE.
- Pueden contener código, datos o recursos (iconos, menus, bitmaps, templates, fonts, etc).
- Proveen modularidad para desarrollo y mantenimiento.

DLL Hell

There are a number of problems commonly encountered with DLLs – especially after numerous applications have been installed and uninstalled on a system. The difficulties include conflicts between DLL versions, difficulty in obtaining required DLLs, and having many unnecessary DLL copies. As a result, an installation of a program that installs a new version of a common object may inadvertently break other programs that were previously installed.

The ambiguity with which DLLs that are not fully qualified can be loaded in the Windows operating system has been exploited by malware in recent years, opening a new class of vulnerability that affects applications from many different software vendors, as well as Windows itself.

Funcionamiento

En Win32 el archivo de las DLL está organizado en secciones:

- Cada sección tiene sus propios atributos (ejecutable, compartible, solo lectura, etc).
  - Generalmente hay una sola copia de las secciones de código en memoria (se comparte).
  - En cambio las secciones de datos son privadas (aunque pueden compartirse).
- Las bibliotecas comprimidas (con UPX por ejemplo) permanecen privadas.
- Los símbolos exportados tienen como identificador un número y un nombre.
  - Solo los nombres se mantienen entre distintas versiones.
  - Se puede hacer un bind a una rutina de una versión específica.

El linkeo puede ser

1. **Load-time dynamic linking:** El linking se hace al cargar el programa. El programador no tiene control si no se encuentra la DLL
2. **Run time dynamic linking:** Se hace por medio de un llamado a `LoadLibrary()`. El programador puede intervenir si hay error.

Para saber que bibliotecas se usan, hay programas como Dependency Walker.

Bibliotecas en Linux Las bibliotecas comienzan con lib. Las de enlace estático terminan en .a y las de enlace dinámico en .so. Por ejemplo la biblioteca foo es libfoo.a y dynfoo es libdynfoo.so.

Además se agrega un número de versión.

Las dependencias se ven con ldd.

Position Independent Code (PIC o PIE)

Es código que puede ejecutar correctamente en forma independiente de su posición en la memoria.

PIC is commonly used for shared libraries, so that the same library code can be loaded in a location in each program address space where it will not overlap any other uses of memory (for example, other shared libraries).

Position-independent code can be executed at any memory address without modification. This differs from relocatable code, where a link editor or program loader modifies a program before execution, so that it can be run only from a particular memory location. Position-independent code must adhere to a specific set of semantics in the source code and compiler support is required. Instructions that refer to specific memory addresses, such as absolute branches, must be replaced with equivalent program counter relative instructions.

Las DLL de Windows no son PIE ni PIC.

Run Time Linking en Linux

Se hace por medio de la interface de programador del linker dinámico: `dlopen()`, `dlsym()`, `dlclose()`, `dlerror()`.

9. Virtualización

Llamamos virtualización a la abstracción de recursos de computación

- Virtualización de Aplicaciones.
- Virtualización de Plataforma.
- Virtualización de Escritorio.
- Virtualización de recursos: Red, Memoria, Almacenamiento, clusters, grids.

Objetivos Permite compartimentar la ejecución de varias aplicaciones en el mismo servidor, de manera independiente y transparente. Independiza la ejecución de la capa física.

Ventajas

- **Aumento de confiabilidad** Si una maquina que alberga muchos servicios se cae, se ven afectados todos. En cambio, si cada servicio está en maquinas distintas, sólo se ve afectado uno. Una maquina física puede tener varias maquinas virtuales (es mas barato que varias maquinas físicas). Sirve porque es (mucho) más común que falle una maquina debido al software que al hardware.
- **Aplicaciones antiguas (legacy)**
- **Desarrollo y prueba en múltiples plataformas**
- **Balanceo de cargas y escalabilidad futura** Es mas fácil migrar de una VM a otra en un host demasiado cargado.

9.1. Virtualizacion de aplicaciones

Compatibilidad y portabilidad entre distintos Sistemas Operativos y distintas arquitecturas.

Application virtualization is software technology that encapsulates application software from the underlying operating system on which it is executed. A fully virtualized application is not installed in the traditional sense, although it is still executed as if it were. The application behaves at runtime like it is directly interfacing with the original operating system and all the resources managed by it, but can be isolated or sandboxed to varying degrees.

Ventajas

- Allows applications to run in environments that do not suit the native application
- May protect the operating system and other applications
- Uses fewer resources than a separate virtual machine (virtualización de plataforma)

Ejemplos

- Máquinas Virtuales (JVM, .net CLR)
- Compatibility Layers

9.2. Virtualización de plataformas

Abstracción de todos los recursos de computación de un huésped dentro de un anfitrión (host). Computer hardware virtualization is the virtualization of computers or operating systems. It hides the physical characteristics of a computing platform from users, instead showing another abstract computing platform. Hace como si fuera otra plataforma en su totalidad.

Tipos de virtualizacion

- Virtualización total
  - Emulación de plataforma.
  - Hipervisores
- Paravirtualización.
- Virtualización del mismo Sistema Operativo.

9.3. Condiciones para virtualizar

Instrucciones privilegiadas Las que ocasionan un software trap.

Instrucciones delicadas ("sensitive") Las que solo pueden ejecutarse en Modo Supervisor del procesador porque afectan a los recursos del sistema, como I/O o cambios en la configuración del MMU.

9.3.1. Teorema de Popek y Goldberg sobre virtualización

Una arquitectura es virtualizable si las instrucciones delicadas son un subconjunto de las privilegiadas. Es decir, si en modo usuario tratar de hacer algo que no deberías poder hacer en modo usuario, debería llamarse un software trap.

9.3.2. Virtualización de la IA32 (x86 virtualization)

x86 tiene 17 instrucciones sensibles que no son privilegiadas, violando una de las condiciones de Popeck por lo que no es virtualizable en estas condiciones.

Para realizar la virtualización de esta máquina se requiere de una extensión que la permita, que en este caso es el Intel IVT, que genera containers en los que la ejecución de una instrucción sensible provoca un software trap. Luego, con la trampa se debe emular el comportamiento de la máquina autónoma del sistema operativo guest.

9.4. Hipervisores

Hipervisor es una pieza de software firmware o hardware que crea y ejecuta máquinas virtuales.

Una computadora en la cuál un hipervisor está ejecutando una o más máquinas virtuales es una maquina "host". Cada máquina virtual es llamada "guest".

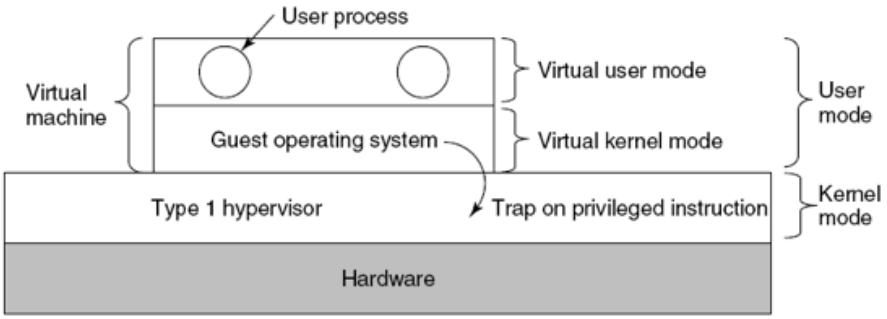
El hipervisor se encarga de la ejecución del sistema operativo del guest, el cuál no sabe que está siendo ejecutado sobre una máquina virtual. Quien está consciente de esto es el hipervisor.

9.4.1. Hipervisor tipo I

Corren directamente sobre el hardware, el huésped debe tener una arquitectura virtualizable.

El huésped corre en modo usuario, su Kernel cree haber pasado a modo supervisor, pero continúa en modo usuario. Cuando el sistema operativo guest ejecuta una instrucción sensible ocurre una "kernel trap".

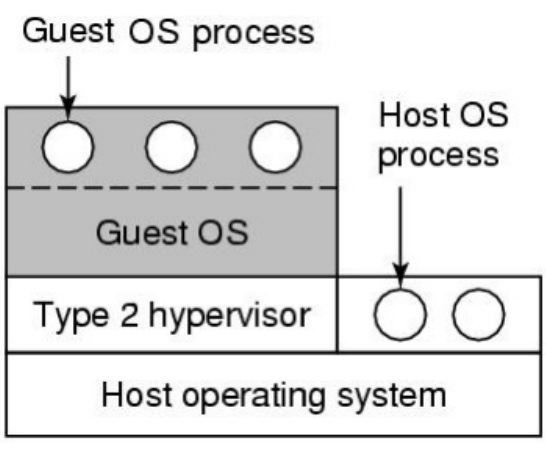
Tiene que ser virtualizable según Popek y Goldberg.



9.4.2. Hipervisor tipo II

Corre bajo el control de un sistema operativo anfitrión o host.

Puede virtualizar cualquier ambiente, aún cuando éste no sea virtualizable según las condiciones de Popek y Goldberg.



9.4.3. Paravirtualización

Paravirtualizar consiste en reemplazar, en el sistema operativo guest, las instrucciones delicadas por llamadas al hipervisor. En este caso, el sistema operativo huésped debe ser modificado para saber que va a correr en un entorno virtualizado. El huésped utiliza una API especial para comunicarse con la capa de virtualización e interactuar directamente con el hard. Si bien tenemos claros beneficios por el lado de la performance, se pierde compatibilidad ya que el SO huésped debe ser modificado. Requiere código fuente.

La paravirtualizacion es distinta del hipervisor tipo II

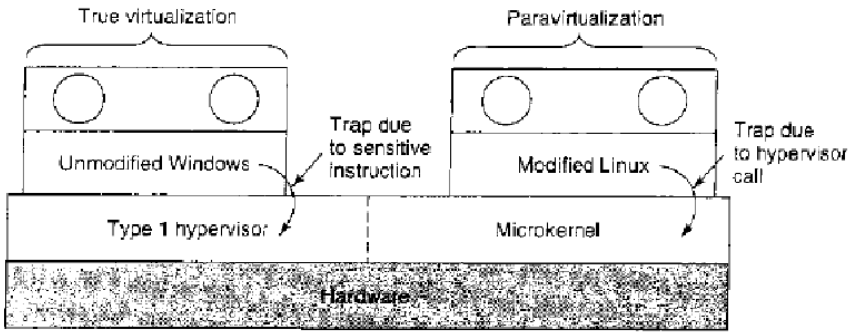


Figure 8-27. A hypervisor supporting both true virtualization and paravirtualization.

9.4.4. Traducción binaria

En tiempo de ejecución cambia las instrucciones sensibles por llamados a procedimientos de VM, es decir modifica la ejecución del programa.

Traducción binaria, en cambio, consiste en examinar bloques básicos (código con un punto de entrada, uno de salida y sin jumps).

El Hipervisor modifica de esta forma el programa que está corriendo. Cuando va a ejecutar un programa, la VM primero examina el código buscando instrucciones sensibles, y las reemplaza por llamados a procedimientos de VM que las manejan.

El Hipervisor toma el control: Si la trap proviene del kernel del guest, lleva a cabo la acción correspondiente, si en cambio proviene de un programa en modo usuario, responde como lo haría el Hard. Guarda el código traducido en el cache, para mejorar la performance.

Esto genera más sobrecarga que el enfoque de paravirtualización, pero no limita a usar sólo SO modificados, podemos usar cualquier SO. No requiere código fuente.

9.5. Virtualización del Escritorio

Las aplicaciones se hospedan en un sistema central pero cada usuario tiene su escritorio local.

Desktop virtualization is software technology that separates the desktop environment and associated application software from the physical client device that is used to access it.

Desktop virtualization can be used in conjunction with application virtualization and (Windows) user profile management systems, now termed "user virtualization", to provide a comprehensive desktop environment management system. In this mode, all the components of the desktop are virtualized, which allows for a highly flexible and much more secure desktop delivery model. In addition, this approach supports a more complete desktop disaster recovery strategy as all components are essentially saved in the data center and backed up through traditional redundant maintenance systems. If a user's device or hardware is lost, the restore is much more straightforward and simple, because basically all the components will be present at login from another device. In addition, because no data is saved to the user's device, if that device is lost, there is much less chance that any critical data can be retrieved and compromised.

9.6. Virtualización de recursos

Usar los recursos del sistema operativo host para apoyar la ejecución del guest.

9.7. Virtualización del Sistema Operativo

Operating system–level virtualization is a server virtualization method where the kernel of an operating system allows for multiple isolated user space instances, instead of just one. Such instances (often called containers, virtualization engines (VE), virtual private servers (VPS) or jails) may look and feel like a real server from the point of view of its owners and users.

Cuando el Kernel permite distintos ambientes de usuarios aislados entre sí.

Usado por seguridad en aplicaciones como hosting virtual.

Virtual hosting environments commonly use operating system–level virtualization, where it is useful for securely allocating finite hardware resources amongst a large number of mutually-distrusting users. System administrators may also use it, to a lesser extent, for consolidating server hardware by moving services on separate hosts into containers on the one server.

Other typical scenarios include separating several applications to separate containers for improved security, hardware independence, and added resource management features. The improved security provided by the use of a chroot mechanism, however, is nowhere near ironclad.

OS-level virtualization implementations that are capable of live migration can be used for dynamic load balancing of containers between nodes in a cluster.

10. Archivos

Bibliografía

[1] Andrew S. Tanenbaum, *Sistemas Operativos Modernos*, tercera edición, PEARSON EDUCACIÓN, México, 2009.

Historial de cambios

21/02/2015 Versión inicial con las secciones de Introduccion, Mecanismos básicos.

22/02/2015 Se agregó la sección de procesos y de threads.