Teoría

Qué es el Sistema Operativo?

El sistema operativo es software, por lo tanto necesita procesador y memoria para ejecutarse. Se compone de un conjunto de instrucciones de alto nivel compilados a máquina, junto a estructuras de datos, para cumplir su objetivo.

Es la capa ubicada entre el hardware y las aplicaciones de usuario. Se llaman clientes, ya que el sistema operativo les ofrece servicios que usan las apps.

Abstrae la arquitectura, ocultándonos el hardware para facilitar el uso de la computadora (friendliness).

Funciones (desde abajo hacia arriba):

- Administra recursos de hardware de uno o mas procesos.
- Da servicios a los usuarios de la computadora o sistema.
- Maneja simultáneamente procesos.
- Multiplexa en tiempo (CPU) y espacio (memoria) los recursos de la máquina.

Objetivos

Los objetivos del sistema operativo son los siguientes:

- Comodidad: hacer fácil el uso de la CPU.
- Eficiencia: hacer uso de forma eficiente de la computadora.
- Evolución: permite introducir nuevas funciones al sistema operativo sin interferir con las ya implementadas. El diseño debe estar apuntado a ello.

Servicios

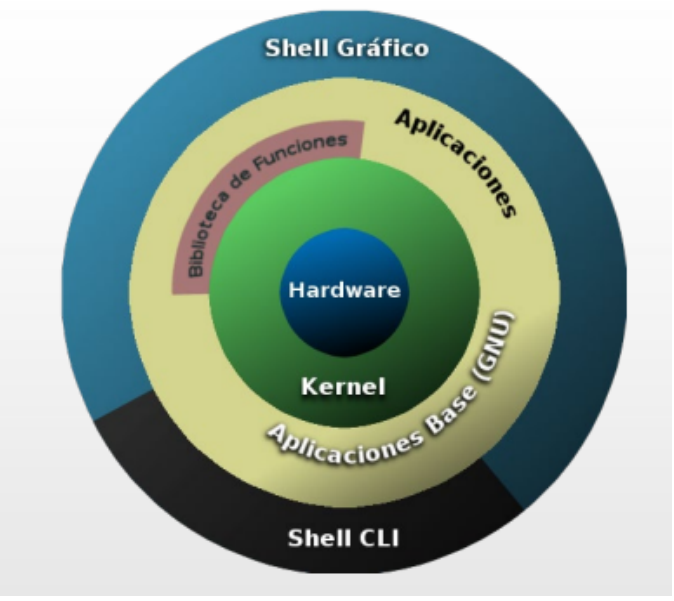
El sistema operativo ofrece una amplia cantidad de servicios ligados a diversas áreas:

- **Administración y planificación de la CPU:**
  - Multiplexa la carga de trabajo
  - Imparcialidad en la ejecución
  - Evitar bloqueos
  - Manejo de prioridades para procesos

- **Memoria:**
  - Administración eficiente
  - Maneja la memoria física y virtual (jerarquía de memoria)
  - Proteje programas que compiten y se ejecutan concurrentemente.
- **Filesystem:** accede a medios de almacenamiento externos
- **Dispositivos:**
  - Ocultamiento de dependencias de HW
  - Administración de accesos simultáneos
- **Detección de errores y respuestas:**
  - Errores de HW: memoria, CPU y dispositivos.
  - Errores de SW: aritméticos y accesos no permitidos
- **Interacción con el usuario (Shell)**
- **Contabilidad**
  - Estadísticas de uso
  - Monitoreo de parámetros de rendimiento
  - Anticipa necesidades de mejoras futuras

Estructura

Los sistemas operativos se pueden dividir en varios componentes. Por un lado está el **Kernel** (núcleo), que está justo por encima del hardware e implementa todo lo necesario para hacer uso de él. Maneja la memoria, CPU, administra procesos, maneja la concurrencia y la entrada y salida. Luego está el **Shell**, que es la forma de interactuar con el sistema operativo (terminal en Linux, ventanas en Windows). El sistema operativo también se compone por **herramientas** como editores, compiladores, librerías y demás.



Problemas a Evitar

Los sistemas operativos deben evitar ciertos problemas que pueden causar los procesos, como por ejemplo:

- Que un proceso se apropie de la CPU
- Que un proceso intente ejecutar instrucciones de entrada o salida privilegiadas.
- Que un proceso intente acceder a una posición de memoria fuera de su lugar asignado → proteger los espacios de direcciones.

Para esto, el SO se debe encargar de gestionar el uso de la CPU, detectar intentos de ejecutar instrucciones de entrada o salida o accesos a memoria ilegales y proteger tanto al vector de interrupciones como a los gestores. El sistema operativo no se puede encargar solo de estas cuestiones, por lo que deberá recibir apoyo del hardware.

Modos de Ejecución

La primera forma de apoyo que nos ofrece el hardware es el modo de ejecución, donde se implementa en la CPU un bit que representa el modo en el que la CPU se está ejecutando. En base al estado de este bit, la CPU opera de una forma u otra.

Modo Kernel

El primer modo de ejecución se llama “**Modo Kernel**” o “Modo Supervisor”. Esto significa que todos los procesos que se intenten ejecutar se podrán llevar a cabo. Sería un “modo superpoder”.

Objetivos:

- **Gestión de procesos:** Crea, termina, planifica, intercambia, sincroniza y ofrece soporte para la comunicación de los procesos.
- **Gestión de memoria:** Reserva espacios de direcciones de memoria para los procesos, para Swapping, Gestión y páginas de segmentos(?).
- **Gestión de E/S:** Buffers, reserva de canales de E/S y de dispositivos de los procesos.
- **Soporte:** gestiona interrupciones, auditoría y monitoreo.

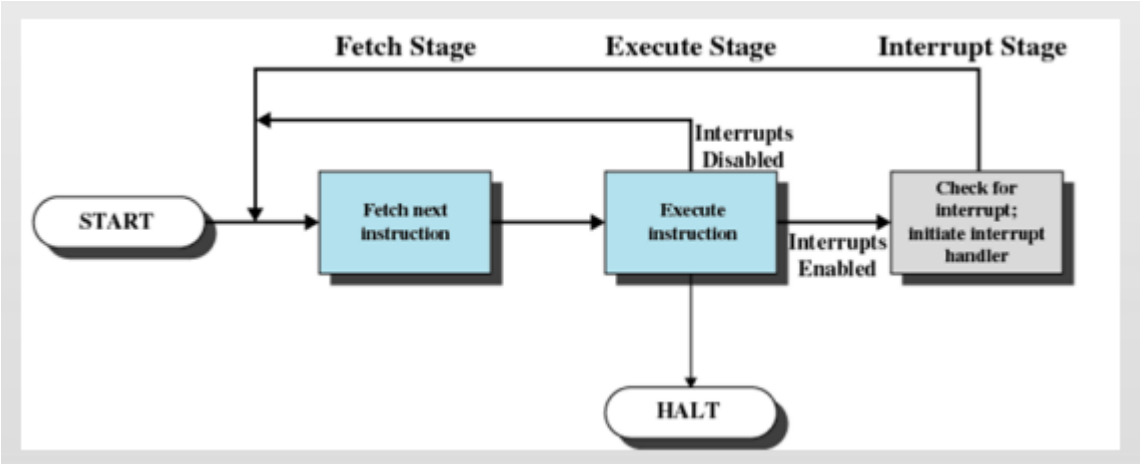
Modo Usuario

El otro modo es el “**Modo Usuario**”, donde se pueden llevar a cabo instrucciones acotadas y acceder a espacios de memoria limitados. En este modo no se puede interactuar con el hardware.

Objetivos:

- Debug de procesos.
- Definición de protocolos de comunicación entre aplicaciones (ej: compilador, editor, apps de usuario).

La forma de alternar entre los modos es con una **interrupción**. Cuando se arranca el SO, este inicia en modo supervisor, y siempre que se inicie un proceso de usuario se debe cambiar el bit a este modo. Cada vez que ocurre una interrupción se pasa al modo kernel.



Cuando queremos realizar el camino inverso (de kernel a usuario), primero se cambia el bit para restringir el alcance de la CPU y luego se le da el control al proceso que se quiere ejecutar.

Si un proceso de usuario intenta ejecutar alguna acción ilegal, el HW lo detecta e interrumpe el proceso.

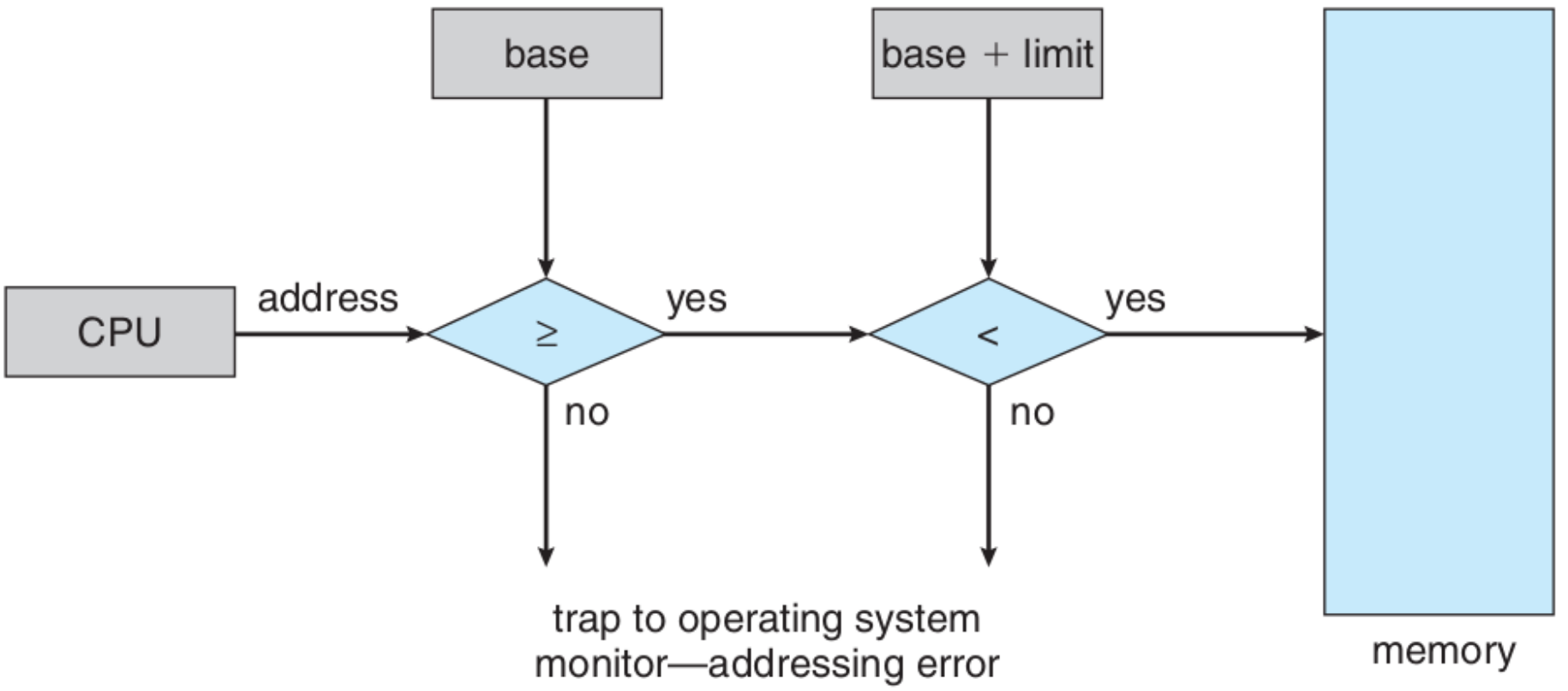
Si estoy en modo kernel y ocurre alguna acción ilegal o un error (división por cero, por ejemplo) se interrumpe por software y se bloquea. Las consecuencias de esto son, por ejemplo, las BSOD (blue screen of death) para reiniciar el sistema.

Interrupción por Clock

Se interrumpe un proceso de usuario con una frecuencia dada por el clock, para evitar que los procesos no se apropien de la CPU. Si hay un proceso que está manejando el CPU durante un tiempo que excede los límites, se da una interrupción y se lo para.

Protección de la Memoria

Cada proceso que se ejecuta tiene un espacio limitado para moverse. Este espacio está limitado entre el registro base y el registro límite.



Si la dirección a la que quiere acceder el proceso de usuario no está dentro del conjunto limitado, se da el trap. De este control se encarga el HW, más precisamente unos registros dentro de la CPU.

System Calls

Los procesos necesitan comunicarse con el sistema operativo para solicitarle servicios (escribir en archivos, cambiar de directorio, etc). La forma de solicitar estos servicios es mediante los system calls.

Estos system calls son procedimientos (o funciones) que llevan consigo una cantidad definida de parámetros, que se pueden pasar de distintas maneras: registros, bloques de memoria o pila.

Los system calls se deben ejecutar en modo kernel, ya que la CPU necesita alcance completo para llevar a cabo la tarea que el proceso no puede.

> 💡 Las direcciones de retorno de las rutinas invocadas durante la atención de un SystemCall son apiladas en el Stack de modo Kernel.

También son interrupciones.

La forma de realizar los system calls es la siguiente:

1. Se llevan los parámetros a la ubicación deseada.
2. Se escribe un número identificador del servicio requerido en un registro determinado.
3. Se llama al proceso que brinda el sistema operativo (servicio)
4. Se fuerza una interrupción por software al proceso de usuario
5. Cambia el modo de ejecución
6. El servicio se ejecuta
7. Volvemos al punto donde se llamó (como cuando volvemos de una interrupción)

La forma de interrumpir es siempre la misma, pero se llama a un servicio o a otro en base al número identificador.

Los system calls cambian de un sistema operativo a otro, por eso un programa hecho para Linux no se puede ejecutar en Windows o MacOS.

Tipos de Kernel

Hay distintos tipos de Kernel, que varían en cuanto a su diseño y funcionamiento:

- **Monolítico:** todas las funcionalidades que debe implementar el sistema operativo se ejecutan en modo kernel, ya que toda la lógica está allí. La ventaja de los monolíticos es que son más rápidos que los microkernels. Windows y Linux son monolíticos.
- **Microkernel:** se intenta hacer el kernel lo más chico posible. En el modo usuario se dejan diferentes componentes que hacen de apoyo al kernel, que queda reducido a sus funciones básicas y esenciales. La filosofía detrás de este tipo de kernel es que mientras menos tiempo se esté en modo kernel, más errores se evitan (por ejemplo, las BSOD).

Procesos

Es un programa en ejecución. Proceso = Job = Tarea.

Programa	**Proceso**
Estático	Dinámico
Sin Program Counter	Con Program Counter
Existe desde que se edita hasta que se borra	Existe desde que se lo dispara hasta que termina

Teniendo un solo CPU, siempre tendremos solo un proceso activo en cualquier instante de tiempo

Componentes de un Proceso

Para poder ejecutarse, un proceso debe contar con las siguientes partes (como mínimo):

- Sección de código.
- Sección de datos.
- Stacks, que son datos temporales como parámetros, variables temporales y direcciones de retorno.

Stacks

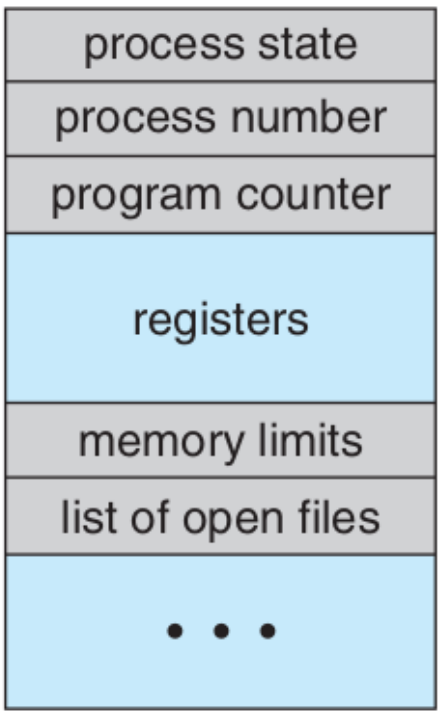
Como mínimo un proceso tiene un stack. Se crean automáticamente y son de característica dinámica, ya que su tamaño varía en run-time acorde a las necesidades del proceso.

Está compuesto por **stack frames**, que contienen parámetros de la rutina y datos necesarios para recuperar el stack frame anterior.

Los stack frames son pusheados a la pila cuando se llama al proceso, y al terminar se poppean.

Process Control Block (PCB)

Todos los procesos tienen un apartado donde se guardan los atributos, los cuales nos dan información acerca de él. Este apartado se llama PCB, que es una estructura de datos que se asocia al proceso, siendo esta lo primero que se crea cuando se lo invoca y lo último que se borra cuando termina.



> 💡 Se dice que un proceso existe desde el momento que se crea su PCB.

Contiene la siguiente información:

- **PID** del proceso.
- **Contexto:** valores de los registros de la CPU (cantidad necesaria para que el proceso pueda continuar en caso de ser interrumpido), estado e información de la memoria.
- **Estado:** ready, running, waiting, etc.
- **Información de Memoria:** dependiendo del sistema de administración de memoria, puede almacenar registro base y límite, tabla de páginas o tabla de segmentos.
- **Información de Planificación:** prioridad, puntero a colas de planificación, …
- Accounting (datos estadísticos para que el SO pueda tomar decisiones).
- E/S (estado, pendientes, archivos abiertos, etc).

Contexto

El contexto de un proceso es toda la info que el SO necesita para administrarlo, y la CPU para ejecutarlo de forma correcta. Se encuentra dentro del PCB.

Los registros de la CPU, el PC, la prioridad de acceso y las operaciones de E/S pendientes son parte del contexto.

Context Switch

Es la operación donde la CPU **cambia el proceso** en estado de ejecución. Para realizar esto, se debe almacenar el contexto del proceso saliente (el cual pasa a espera). Luego se carga el contexto del proceso que ingresa, para comenzar desde la instrucción siguiente a la última ejecutada.

El context switch es tiempo no productivo para la CPU, y la cantidad de ciclos que consuma depende del soporte del HW.

Espacio de Direcciones

Es el conjunto de direcciones de memoria que ocupa un proceso, a las cuales puede acceder.

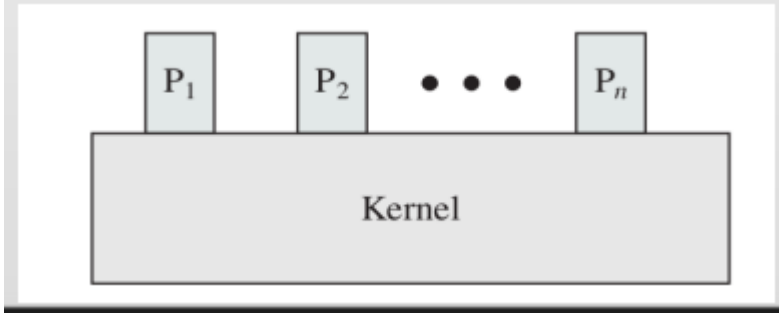
Este espacio no incluye al PCB. Sin embargo, si estamos en modo Kernel, se pueden acceder a estructuras internas (incluido el PCB) o a espacios de direcciones de otros procesos.

El Kernel como Proceso

Al ser un conjunto de módulos de software que se ejecuta en el procesador, tendría sentido que fuera un proceso. Sin embargo, no lo es, y hay dos formas de verlo:

Entidad Independiente

El kernel es una porción de código que se ejecuta por fuera de todo proceso como una identidad independiente, y en modo privilegiado (los procesos se asocian a programas de usuario).



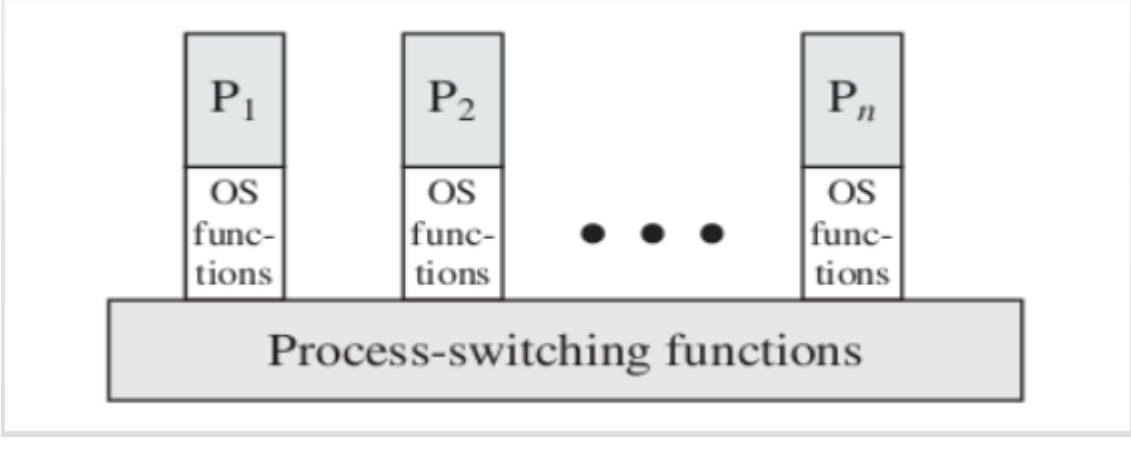
Cuando un proceso es interrumpido o realiza un System Call, se resguarda su contexto y se le otorga el control al Kernel del sistema.

El Kernel también cuenta con región de memoria y stack propios, y una vez que termina lo que tenía que hacer, le devuelve el control al proceso que se estaba ejecutando o a otro diferente.

Kernel Dentro del Proceso

Podemos pensar que el Kernel está dentro de los procesos ya que su código se encuentra, literalmente, dentro del espacio de direcciones de cada proceso, y se ejecuta en el mismo contexto que algunos procesos de usuario.

Además, podemos ver al Kernel como una serie de rutinas que los procesos de usuario utilizan.



Como podemos ver, dentro de un proceso se encuentra el código del programa (user) y el de los módulos del SW del SO (Kernel). Por ello, cada proceso cuenta con un stack en modo usuario y otro stack en modo Kernel.

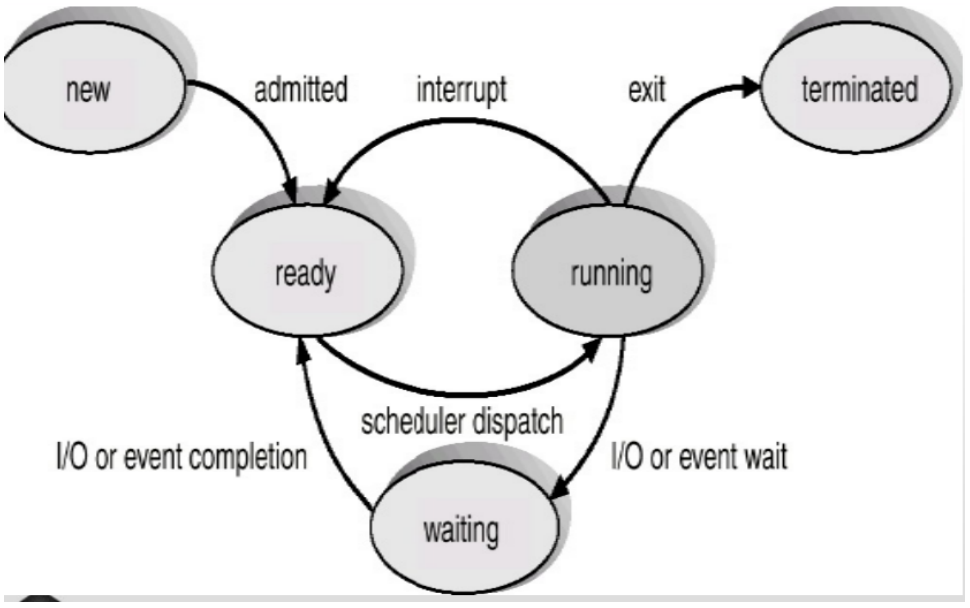
El proceso es el que se ejecuta en modo Usuario, y el Kernel del sistema operativo es el que se ejecuta en el modo Kernel.

Entonces el código del kernel es compartido por todos los procesos.

Cada interrupción que se dé durante la ejecución de un proceso será atendida en el contexto del mismo, pero en modo kernel (hay un cambio de modo sin necesidad de realizar un cambio completo de contexto).

El SO determina si el proceso deberá seguir ejecutándose luego de atender la interrupción. Si es así, cambia al modo usuario y devuelve el control.

Estados de un Proceso



- **New**: es el estado en el que aparece un proceso recién inicializado por el padre, y donde se crean todas las estructuras necesarias para que se ejecute. Luego de que se creen estas estructuras, quedará en la cola de procesos esperando a ser cargado en memoria.
- **Ready**: el long term elige al proceso para cargarlo en memoria, y pasa a estado ready. Ahora lo único que le falta para que se ejecute es que se le asigne CPU. Está en la ready queue.
- **Running**: el proceso se está ejecutando gracias a que el short term lo eligió y hubo un context switch. Va a tener CPU hasta que se le termine el tiempo asignado o necesite realizar una E/S.
- **Waiting**: estado donde el proceso espera una operación de entrada/salida o una señal de otro proceso para volver a competir por la CPU, pero sigue estando en la memoria principal. Cuando lo que está esperando ocurre, vuelve al estado ready.
- **Terminated**: se eliminan las estructuras creadas en memoria para que el proceso haya podido correr.

💡 Cuando un proceso está en estado **running** y se le acaba el quantum, es expulsado de la CPU. Esto es un caso especial, ya que normalmente sale porque necesita que se complete algún evento externo.

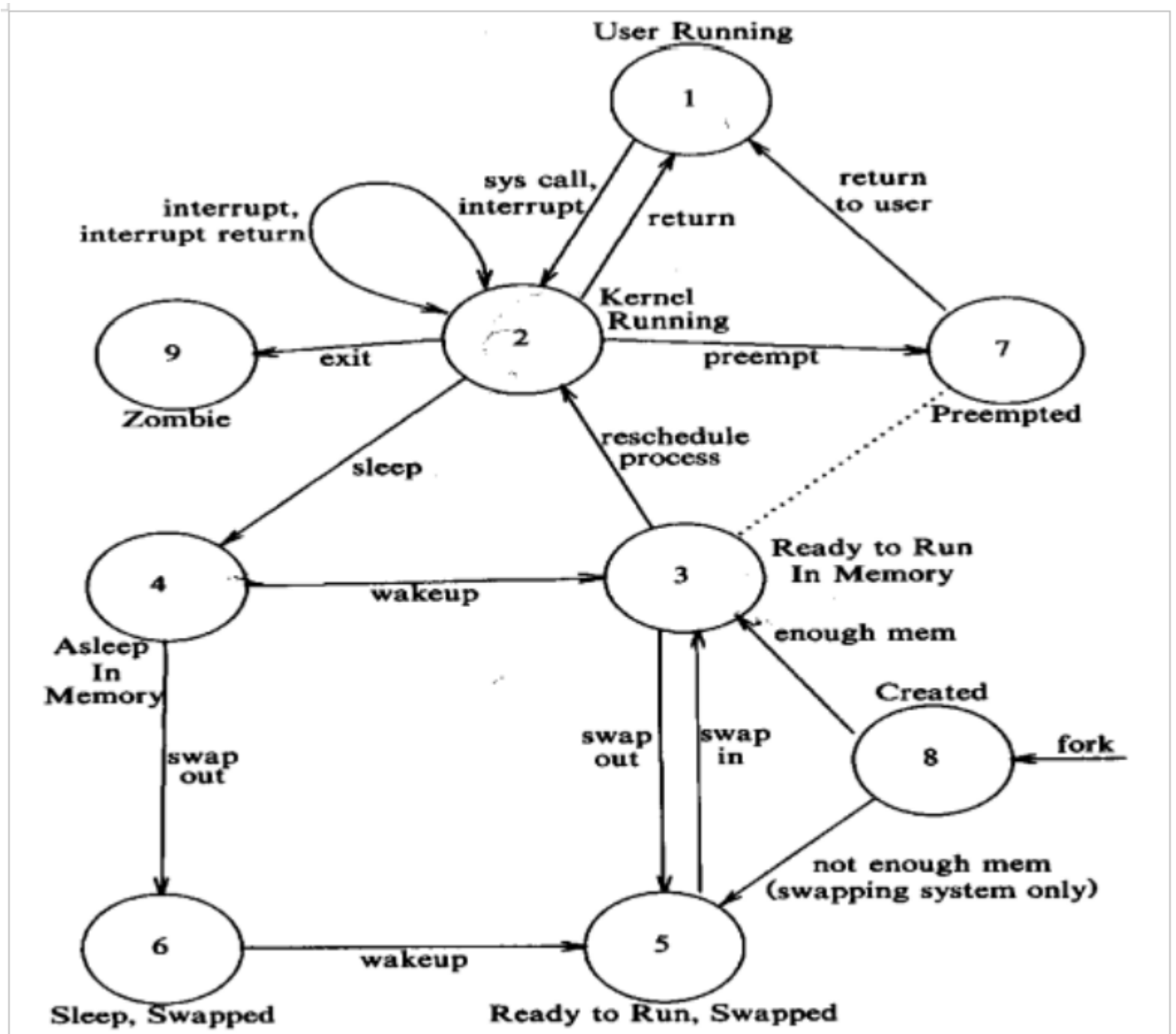
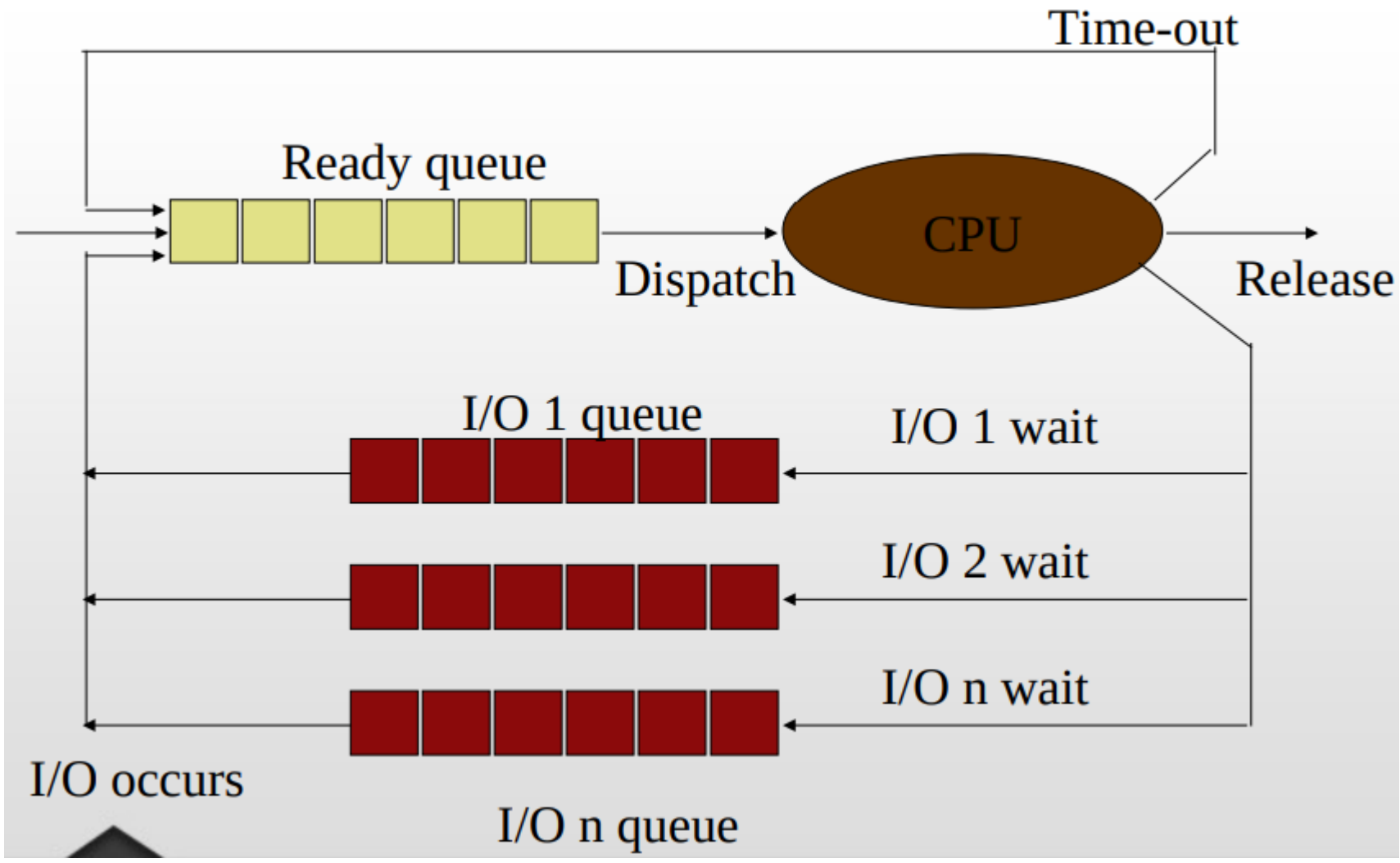


Figure 6.1. Process State Transition Diagram

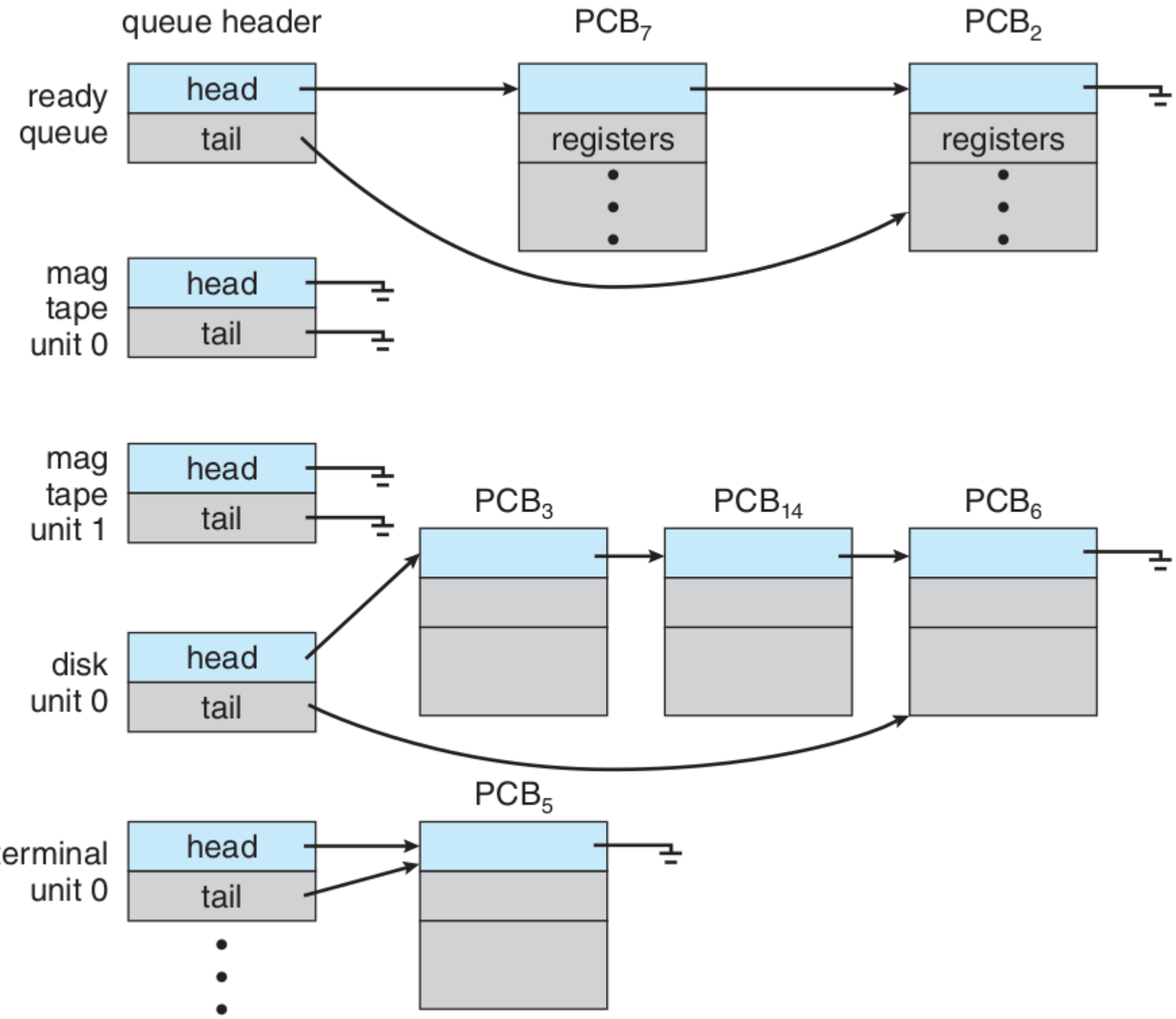
💡 Los procesos nacen y mueren en memoria principal, lo que se swapea es su contenido.

Colas de Planificaciónn

El sistema operativo guarda una o más colas de planificación para cada estado de proceso. Pueden haber muchos procesos en el mismo estado a la vez (excepto en running, si contamos con una CPU).



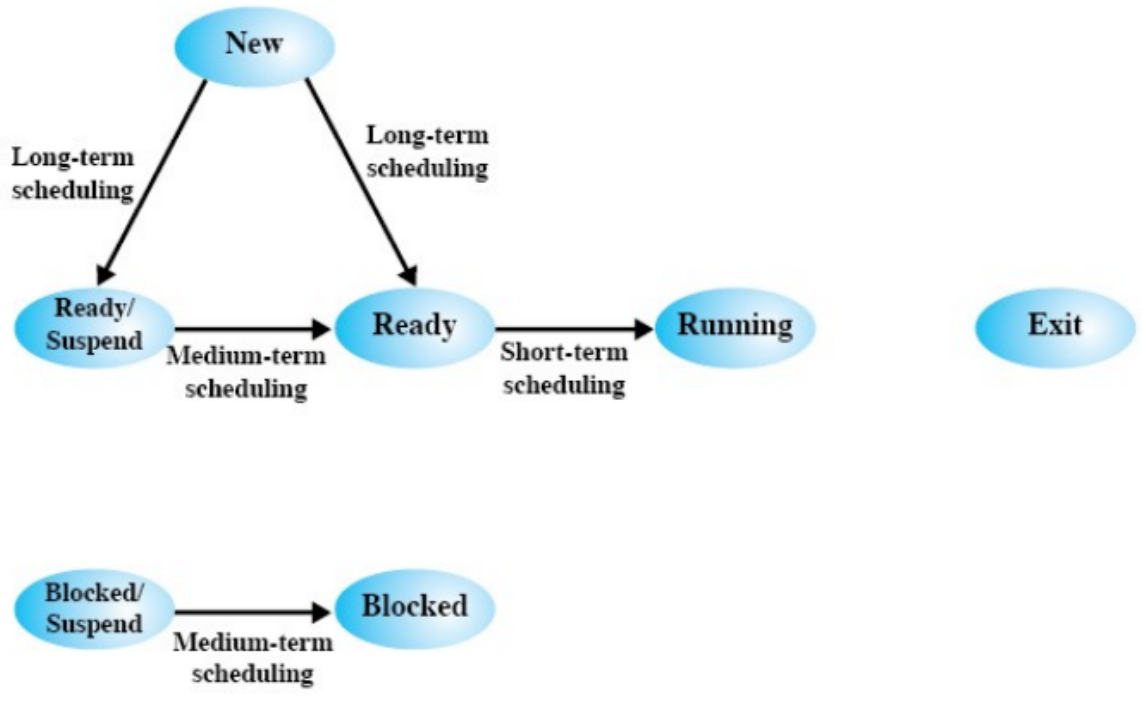
Las colas no almacenan los procesos en sí, sino que su PCB (abstracción del proceso), los cuales son enlazados siguiendo un orden determinado (no cambian de ubicación en la memoria, cambia el sig y el ant).



Las colas pueden tener algoritmos de planificación distintos, no todas se manejan con el FIFO.

Módulos de Planificación

Son una parte del código del Kernel que se encargan de planificar procesos. No se están ejecutando constantemente, sino que se disparan ante determinados eventos, como lo son la creación o terminación de procesos, entrada o salida, etc.



En donde se ve involucrado cada scheduler

- **Long Term Scheduler:** es el planificador que se encarga de decidir, de todos los procesos creados en estado new, cuál es admitido en memoria para pasar al estado ready. Controla el grado de multiprogramación.
- **Short Term Scheduler:** selecciona entre los procesos en estado ready cuál toma la CPU para pasar a running.
- **Medium Term Scheduler:** reduce el grado de multiprogramación cuando sea necesario, sacando procesos de la memoria. Esto lo lleva a cabo mediante el *swap in* (meter un proceso a memoria) y *swap out* (sacarlo).

> 💡 El grado de multiprogramación es la cantidad de procesos en memoria. Si es estable, el número de procesos creados debe ser similar al de procesos terminados.

Short Term	Medium Term	Long Term
Selecciona entre los procesos en estado ready cuál toma la CPU para pasar a running.	Controla el grado de multiprogramación mediante el *swap in* y *swap out*.	Decide de entre todos los procesos creados en estado new cuál es admitido en memoria para pasar al estado ready.
Ready → Running	**Suspend → Ready Suspend → Blocked**	**New → Ready New → Suspend**

Además de los planificadores anteriores también hay otros dos módulos, los cuales podrían no existir como planificadores separados, pero su función se debe cumplir.

- **Dispatcher:** se encarga de realizar el context switch. Despacha el proceso elegido por el short term.
- **Loader:** carga en memoria el proceso elegido por el long term.

Algoritmos de Planificación

A gran escala, se pueden dividir en dos:

- **Apropiativos:** preemptive. El proceso en ejecución puede ser expulsado de la CPU.
- **No Apropiativos:** nonpreemptive. Los procesos se ejecutan hasta que abandonen la CPU por su cuenta.

Los sistemas donde trabajan los algoritmos de planificación tienen distintas metas, y los algoritmos deben ajustarse a ellas. Esta meta puede ser otorgar una parte justa de la CPU a cada proceso (equidad), o podríamos querer mantener ocupadas todas las partes del sistema (balance).

Procesos Batch

Son procesos que se ejecutan completamente en el background, sin interacción con el usuario, por lo tanto son **CPU bound**.

Los procesos batch tienen como meta el rendimiento (más trabajos en menos tiempo), minimizar el tiempo de retorno, y mantener la CPU ocupada el mayor tiempo posible.

Es común implementar los batch con algoritmos no apropiativos, como por ejemplo FCFS (first come first served) o STF (shortest time first).

Procesos Interactivos

Interactúan principalmente con el usuario, pero también con servidores (**I/O bound**). Para ofrecer una mayor interactividad necesitamos de algoritmos apropiativos, para que ningún proceso se quede mucho tiempo en la CPU.

Los objetivos de estos procesos son ofrecer un tiempo de respuesta más corto respondiendo a las peticiones rápidamente, y proporcionalidad, es decir, cumplir con las expectativas de los usuarios.

> 💡 Hay una necesidad de atender rápidamente los procesos I-O Bound para mantener los dispositivos ocupados y aprovechas más la CPU para procesos CPU-bound.

Los algoritmos apropiativos que se suelen implementar son Round Robin, prioridades, colas multinivel y SRTF (shortest remaining time first).

Política/Mecanismo

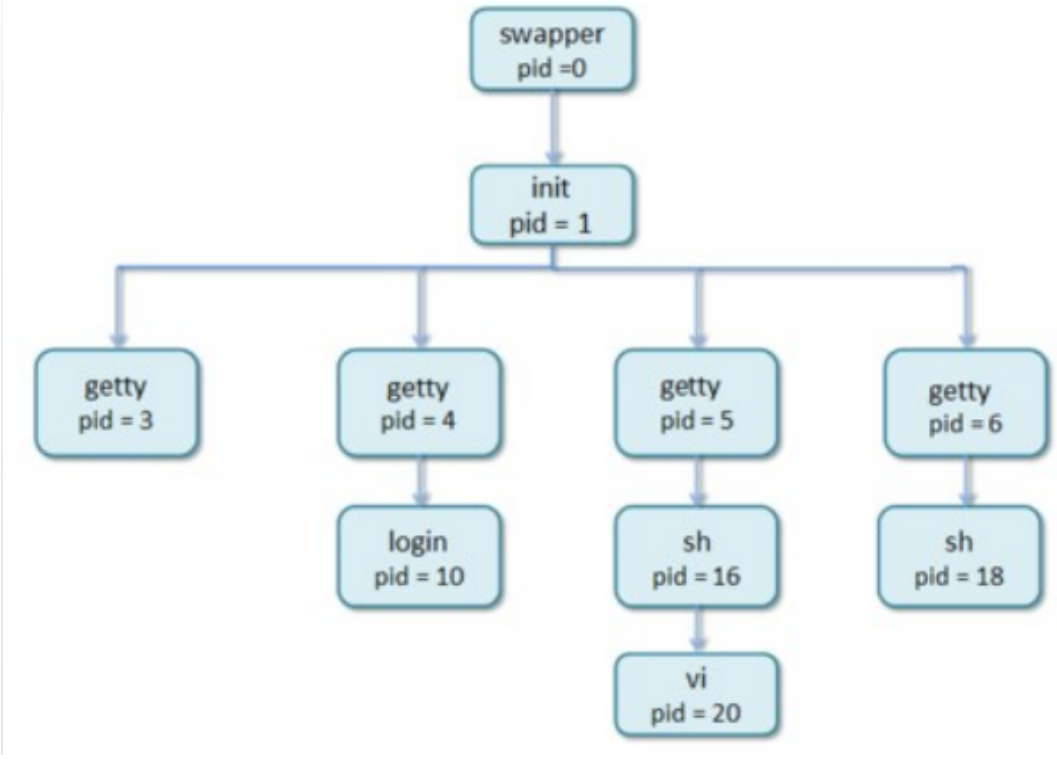
El Kernel del SO implementa el mecanismo (algoritmo) de planificación, que puede ser RR, o cualquiera de los que vimos. Dependiendo del fin que se le de al sistema, la política de cómo se aplica ese algoritmo será distinta.

> 💡 Por ejemplo, si tenemos una máquina que actúa como servidor, no tiene sentido poner un quantum chico, ya que no ejecutará procesos de entrada/salida. Lo mismo para computadoras personales: si le ponemos un quantum grande la computadora no va a ser lo suficientemente interactiva.

Según el ambiente, podríamos llegar a requerir algoritmos de planificación diferentes en base al fin de nuestro SO. Podemos buscar que todos los procesos tengan una parte justa de la CPU (**Equidad**) o mantener ocupadas todas las partes del sistema (**Balance**).

Creación de Procesos

Los procesos se van creando en forma de árbol (pueden tener uno o más procesos hijos).



La creación de un proceso conlleva los siguientes pasos:

1. Creación del PCB.
2. Asignación de PID.
3. Memoria a las regiones Stack, Text y Datos.
4. Estructuras de datos asociadas (Fork): copiar contexto y regiones.

Cuando un proceso padre crea un proceso hijo, por lo general continúan ejecutándose concurrentemente o compitiendo con la CPU. Otra cosa que puede pasar es que el padre espere a que el hijo termine para seguir ejecutándose, pero es obsoleto.

Cuando se crea un hijo, su espacio de direcciones es un duplicado del padre (hereda espacio de direcciones). El contenido no, sino que las direcciones lógicas.

> 💡 **fork()** crea un nuevo proceso y **execve()** carga un nuevo programa en el espacio de direcciones.
>
> En Windows **CreateProcess()** crea un nuevo proceso y carga el programa para ejecución.

Fork

Se utiliza para crear un nuevo proceso.

- Si tuvo éxito → retorna el PID del proceso creado (int > 0).
- Si estoy en el hijo → retorna 0.
- Error → retorna negativo (int < 0).

Terminación de Procesos

Si ocurre un `exit` , se retorna el control al sistema operativo. El padre puede esperar a que el hijo le mande un código de retorno (`await`) para continuar.

El padre también puede terminar la ejecución de los hijos con `kill`. Esto ocurre cuando la tarea que el hijo tenía que realizar es completada o cuando el padre deja de ser ejecutado (terminación en cascada).

Procesos Cooperativos e Independientes

Hay problemas que pueden ser solucionados de forma más eficaz de forma concurrente y paralela.

Independencia: el proceso no afecta ni es afectado por otros. Tampoco comparte ningún tipo de dato.

Cooperativo: afecta o es afectado por la ejecución de otros procesos.

Para implementar procesos cooperativos es necesario que el SO nos permita compartir información entre procesos mediante herramientas que nos provea. También es necesario poder planificar en paralelo.

Los procesos en cooperativo nos permiten acelerar el cómputo.

Memoria

El sistema operativo maneja la memoria abstrayendo los conjuntos de celdas a espacios de direcciones lógicos. Para unir esta abstracción con la memoria física, el SO cuenta con soporte del HW.

Los programas deben estar en la memoria principal para poder referenciarlos y ejecutarlos.

Como funciones principales, el sistema operativo debe:

- Llevar un registro de las porciones de memoria utilizadas y no utilizadas.
- Asignar memoria principal a los procesos cuando la necesiten.
- Liberar espacio de memoria asignada a procesos que ya terminaron.
- Abstraer la alocación física de los programas al programador.
- Evitar que un proceso entre al espacio de direcciones de otro proceso.
- Brindar acceso compartido a determinadas secciones de memoria.

Además, el SO debe ser eficiente en cuanto al uso de la memoria para tener una mayor cantidad de procesos almacenados a la vez. Esta administración depende del mecanismo que provea el HW.

Otra tarea que también le corresponde al Hardware (registros de la CPU) es el control de accesos indebidos a memoria. Se detecta mediante el código de instrucción, y cuando ocurre el HW produce un *trap* al sistema operativo.

Requisitos

Como mínimo, el sistema operativo debe proveer un cierto estándar de funciones relacionadas a la memoria. Todas estas van con apoyo del HW:

- **Reubicación:** la memoria debe ser capaz de reubicar procesos que estaban en ejecución, llevándolos y trayéndolos (swap). Las direcciones de memoria se deben traducir según la ubicación actual del proceso. El programador no debe tener la necesidad de saber dónde se encuentra el proceso en la memoria.
- **Protección:** los procesos no deben acceder a las direcciones de memoria de otros procesos, salvo que tengan permiso de hacerlo. Este chequeo se realiza durante la ejecución, pero es imposible anticipar todas las referencias a memoria que un proceso vaya a realizar.
- **Compartición:** permitir que los procesos tengan la posibilidad de compartir datos para trabajar en común, y evitar copias innecesarias de información.

Abstracción

Es el rango de direcciones lógicas posibles que un proceso puede utilizar para direccionar datos e instrucciones. Es independiente de la ubicación física del proceso en la memoria RAM.

El tamaño depende de la arquitectura del procesador:

- 32 bits: 0 .. 2^32.
- 64 bits: 0 .. 2^64.

Direcciones

Como dijimos, hay dos tipos de direcciones.

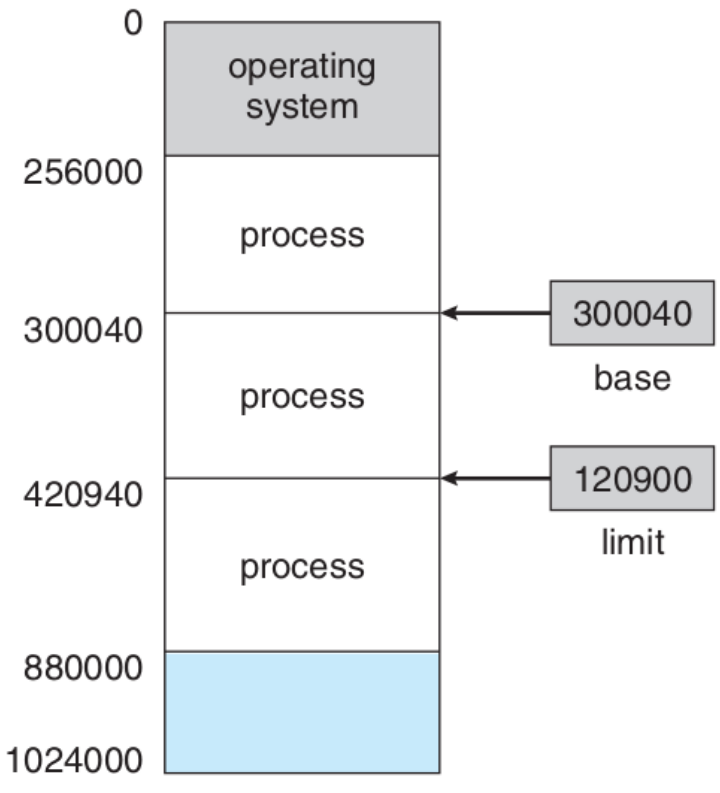
Por un lado tenemos las **lógicas**, que hacen referencia a una localidad de memoria independiente de la asignación física. Representa una dirección dentro del espacio de direcciones del proceso, es una dirección relativa.

Por otro lado están las direcciones **físicas**, que representan una localidad concreta en la memoria RAM. Se le llama dirección absoluta.

Conversión de Direcciones

Para que el sistema pueda funcionar de forma correcta y el HW hable con el SO, es necesario que las direcciones de memoria se traduzcan de lógicas a físicas, y viceversa. Una forma de hacer esto es mediante los registros auxiliares:

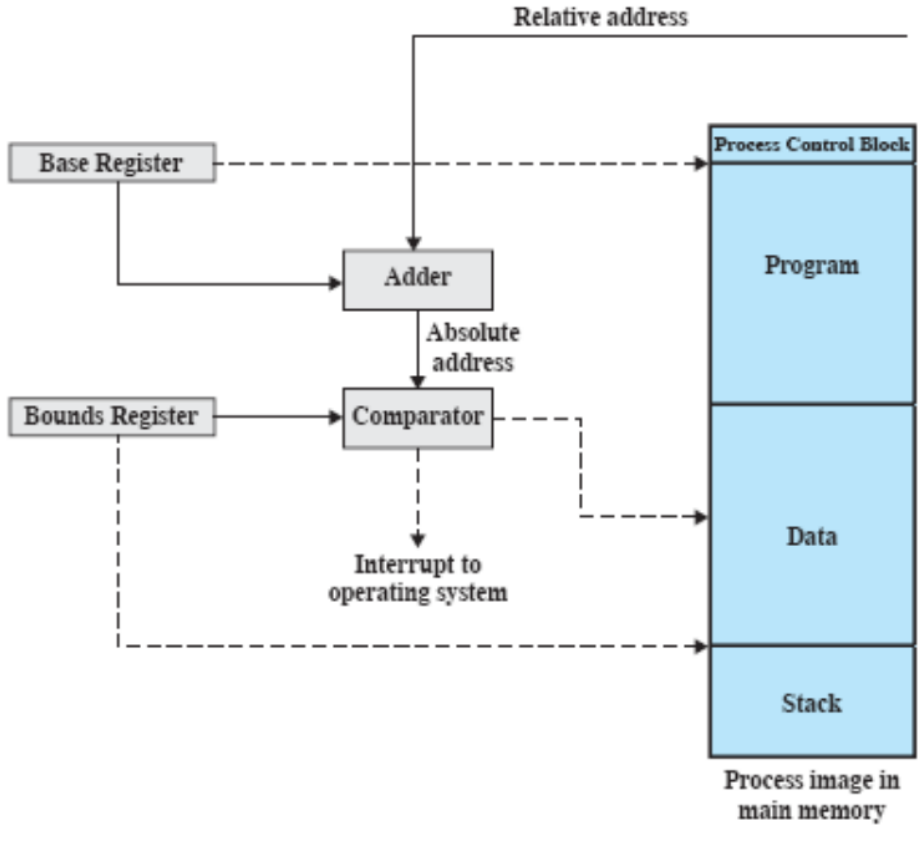
- **Registro Base:** dirección física donde comienza el espacio de direcciones del proceso en la RAM.
- **Registro Límite:** dirección donde finaliza el espacio del proceso, o también puede ser cuántas direcciones abarca desde el registro base.



La CPU trabaja con direcciones lógicas, por lo que para acceder a memoria principal se le deben traducir las físicas.

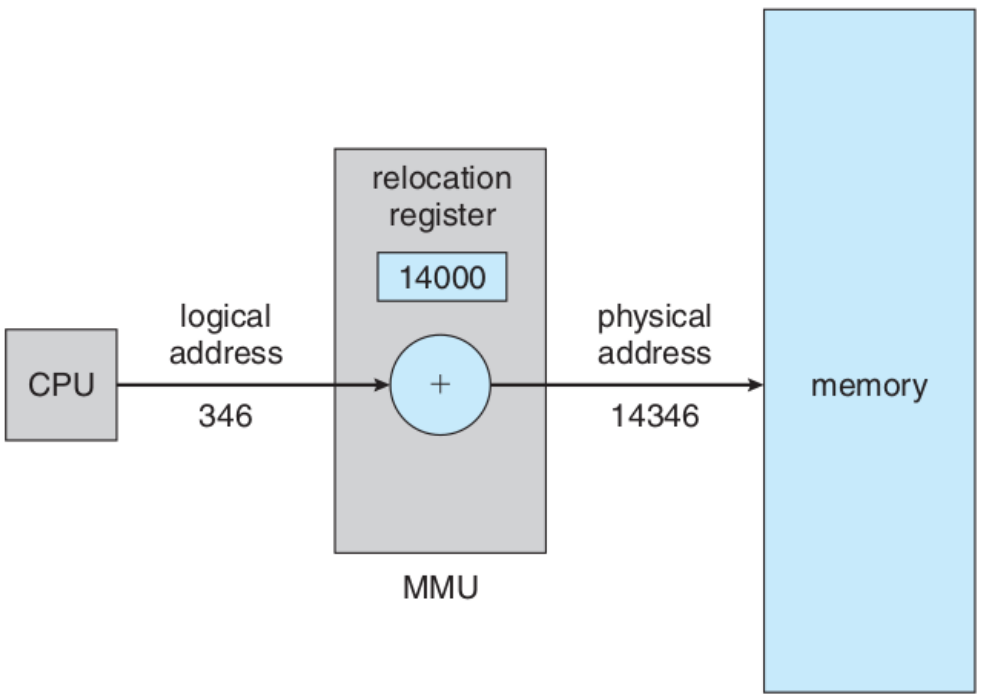
> 💡 El proceso de resolución de direcciones se llama **address-binding.**

El address-binding se realiza en tiempo de ejecución, y funciona por hardware. El binding debe ir a la velocidad del procesador (rápido), por lo que se asigna una unidad específica para esta tarea: **Memory Management Unit (MMU).**



Memory Management Unit

Es una parte del procesador que traduce de direcciones virtuales a físicas y de físicas a virtuales. La operación de traducción se realiza con permisos privilegiados, en Kernel Mode.

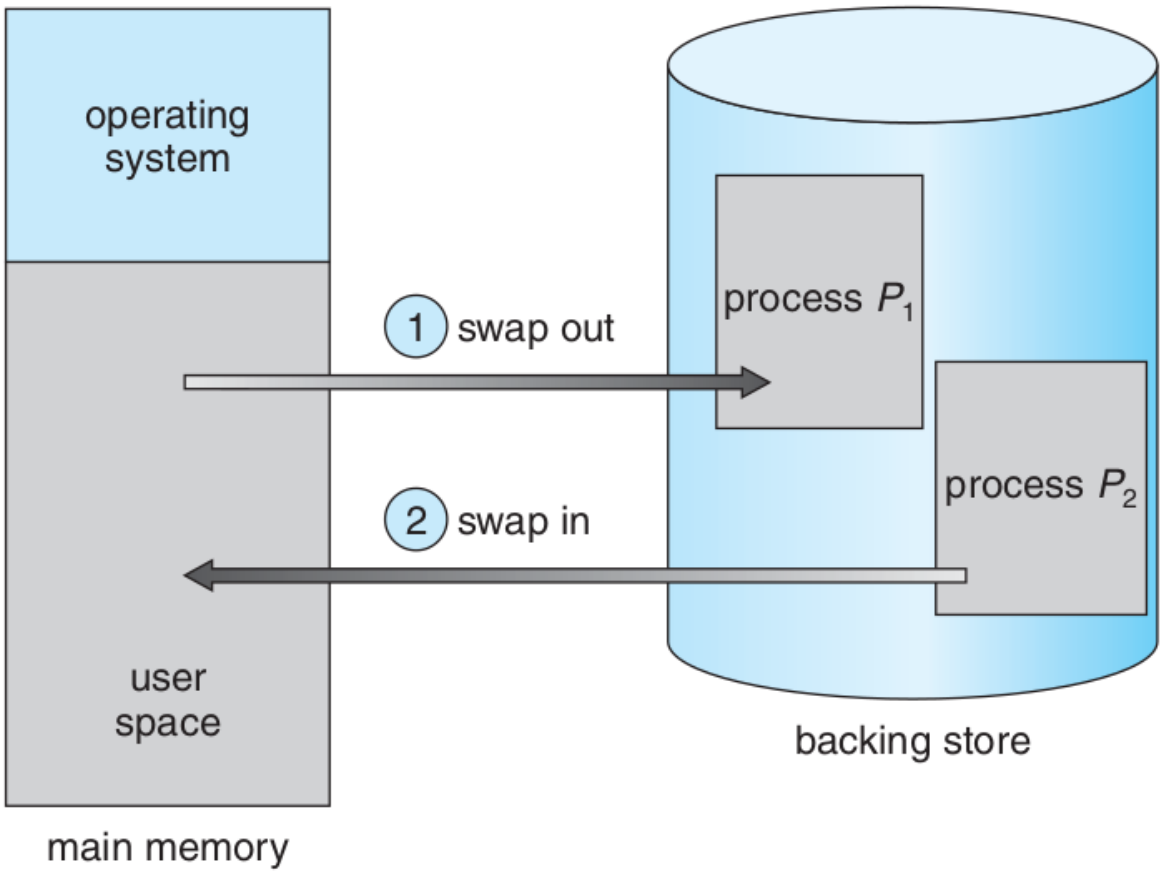


Tiene un valor en el registro de realocación, que se le suma a la dirección virtual generada por el proceso de usuario cuando quiere acceder a memoria.

Los procesos de usuario nunca usan direcciones físicas.

Swapping

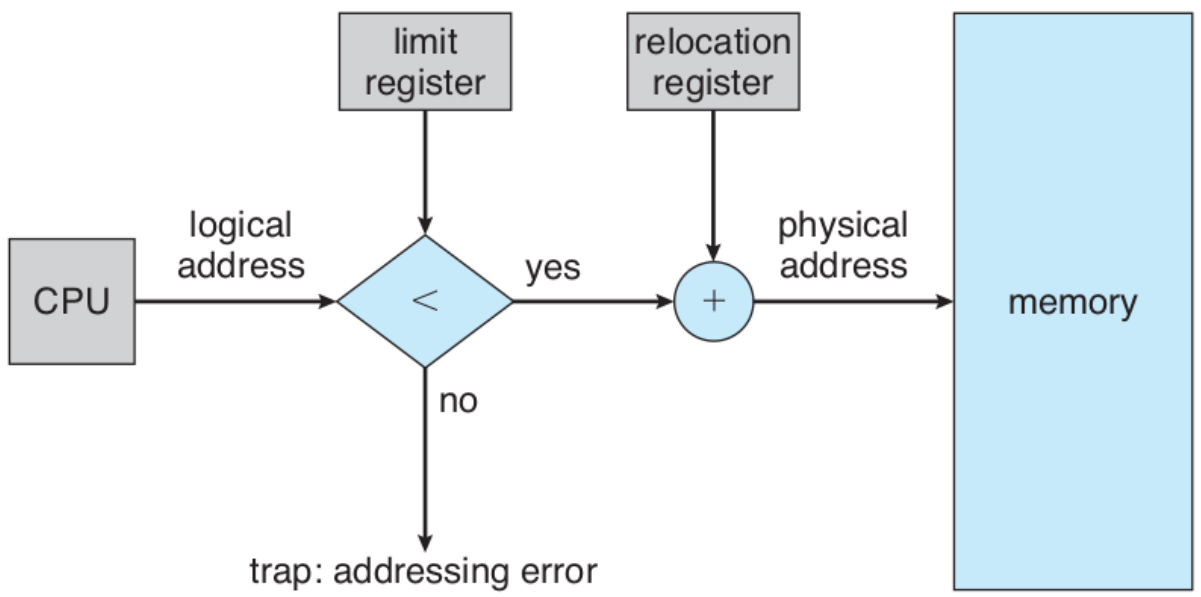
Es la operación donde se llevan o traen procesos desde memoria principal a memoria secundaria, o de memoria secundaria a principal. El swapping es necesario ya que hace posible que el tamaño total de los procesos excedan el tamaño total de la memoria principal, teniendo el apoyo de memoria virtual.



Alocación Continua

En los primeros sistemas operativos, la asignación de memoria se hacía mediante particiones. Esto era así debido a que el hardware venía configurado de esta forma, y el SO se adaptaba a ello.

La alocación de memoria era contigua, es decir que los procesos estaban alojados en una sección única de memoria, y junto a esa sección había otro proceso.



Asignación por Particiones

Esquema de alocación continua donde la memoria se divide en particiones, y cada partición almacena exactamente un proceso. Cuando un proceso quiere acceder a memoria, debe fijarse que hayan particiones disponibles. Si hay, entra. Sino, espera a que algún proceso ocupando memoria termine.

- **Particiones Fijas:** la memoria se dividía en regiones de tamaño fijo (todas del mismo tamaño o no), donde cada una alojaba un proceso, que se colocaba allí de acuerdo a algún criterio.
- **Particiones Dinámicas:** las particiones pueden variar en tamaño y cantidad. Siguen alojando un proceso cada una, pero su tamaño se ajusta a la necesidad del proceso.

Fits

Son los criterios implementados para decidir en qué agujero (partición libre) va un proceso.

- **First Fit:** asigna el primer bloque de memoria que es lo suficientemente grande para el proceso.

- **Best Fit:** busca el bloque de memoria más pequeño que sea suficiente para el proceso y lo asigna. Se debe buscar en toda la memoria, pero es la que deja el agujero más pequeño.
- **Worst Fit:** Asigna el bloque de memoria más grande disponible que sea suficiente para el proceso. Es la que más espacio desperdicia.
- **Next Fit:** Similar al first fit, pero comienza la búsqueda en el lugar donde se quedo la búsqueda anterior en lugar de comenzar desde el principio

Fragmentación

La **fragmentación** se da cuando hay espacio de memoria libre suficiente para alojar un proceso, pero no podemos utilizarla.

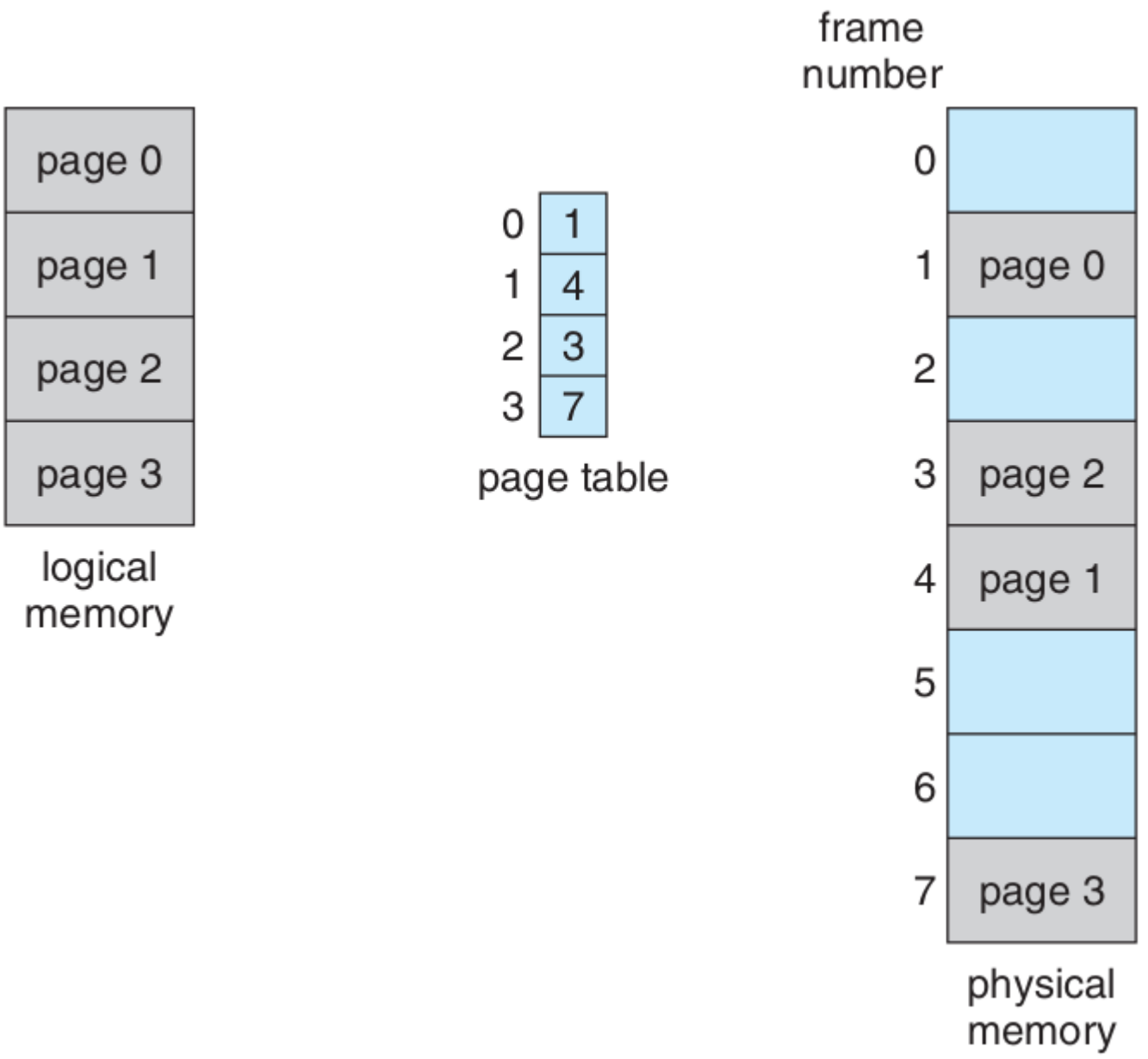
- **Fragmentación Interna:** se produce en esquemas de asignación donde los tamaños de los bloques asignados no cambian. Hay una porción de bloque que no se utiliza, cuyo espacio es desperdiciado y no puede asignarse a otro proceso ya que el tamaño no cambia.
- **Fragmentación Externa:** se produce en esquemas de bloques dinámicos. Son los huecos que van quedando en la memoria a medida que los procesos finalizan. Como no se encuentran de forma contigua, puede ocurrir que tengamos memoria libre para alocar un proceso, pero que no se pueda utilizar. Una solución a esto es compactar la memoria, para unir los huecos y dejarlos todos juntos. Esto es muy costoso en tiempo y recursos.

Paginación

A medida que comenzaron a surgir problemas con el esquema de registro base + límite, como por ejemplo la necesidad de almacenar el espacio de direcciones de forma continua en la memoria (porque sino tenemos la fragmentación, que tarde o temprano va a aparecer), surge un nuevo esquema: el de la paginación.

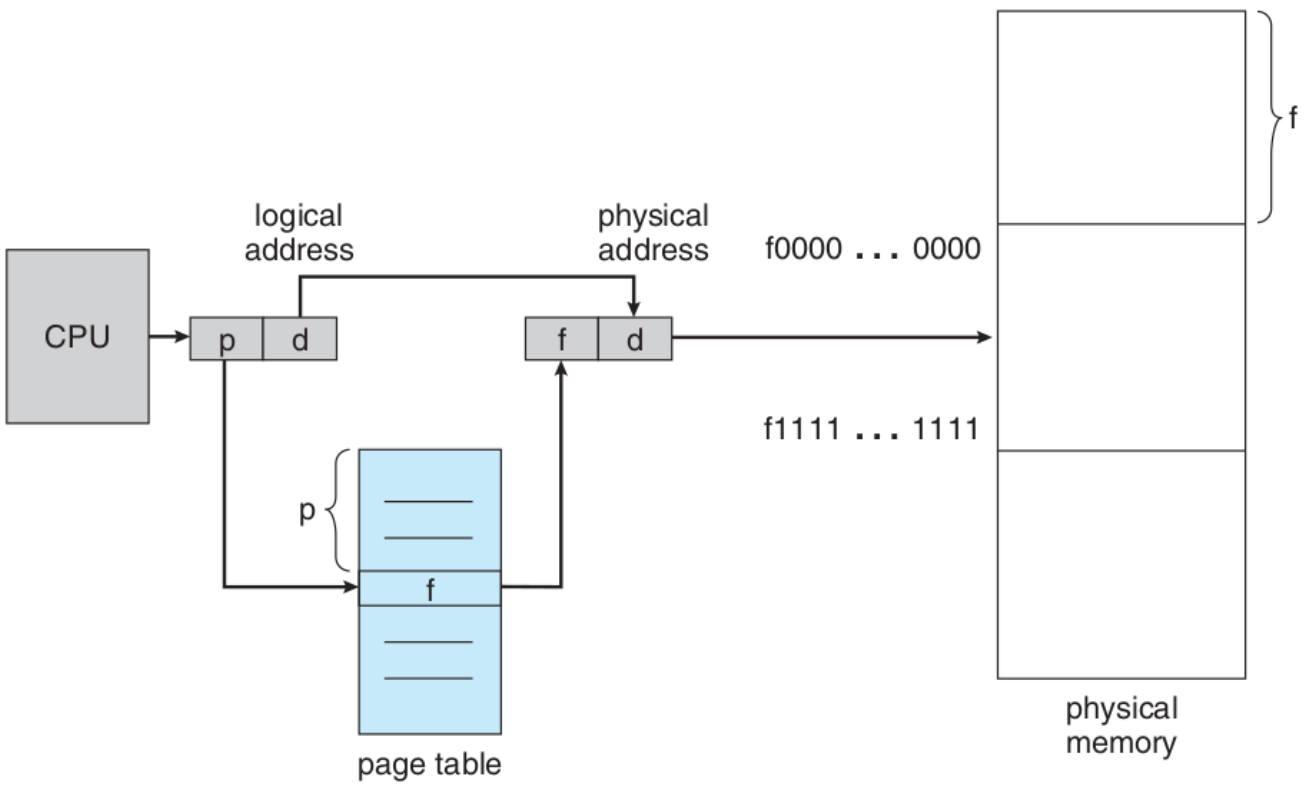
Consiste en dividir lógicamente la memoria física en pequeñas partes iguales (llamadas **marcos**), y dividir la memoria lógica en partes de igual tamaño que los marcos (**páginas**). Bajo esta división, cualquier página puede ir a parar a cualquier marco, solucionando el tema de la continuidad.

Generalmente, el tamaño de las páginas es de 512 Bytes.



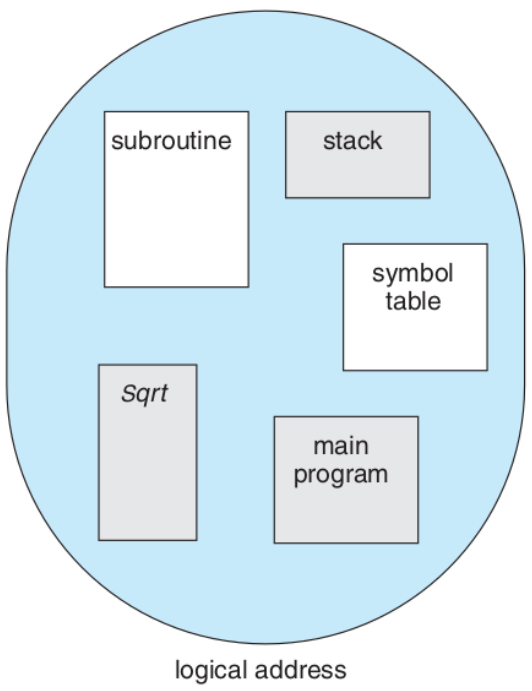
El sistema operativo debe saber mediante un registro en qué marco está cada página. A este registro se lo llama **tabla de páginas**.

Las direcciones lógicas ahora se interpretan como un número de página, sumado a un desplazamiento dentro de la misma: v = (p, d).

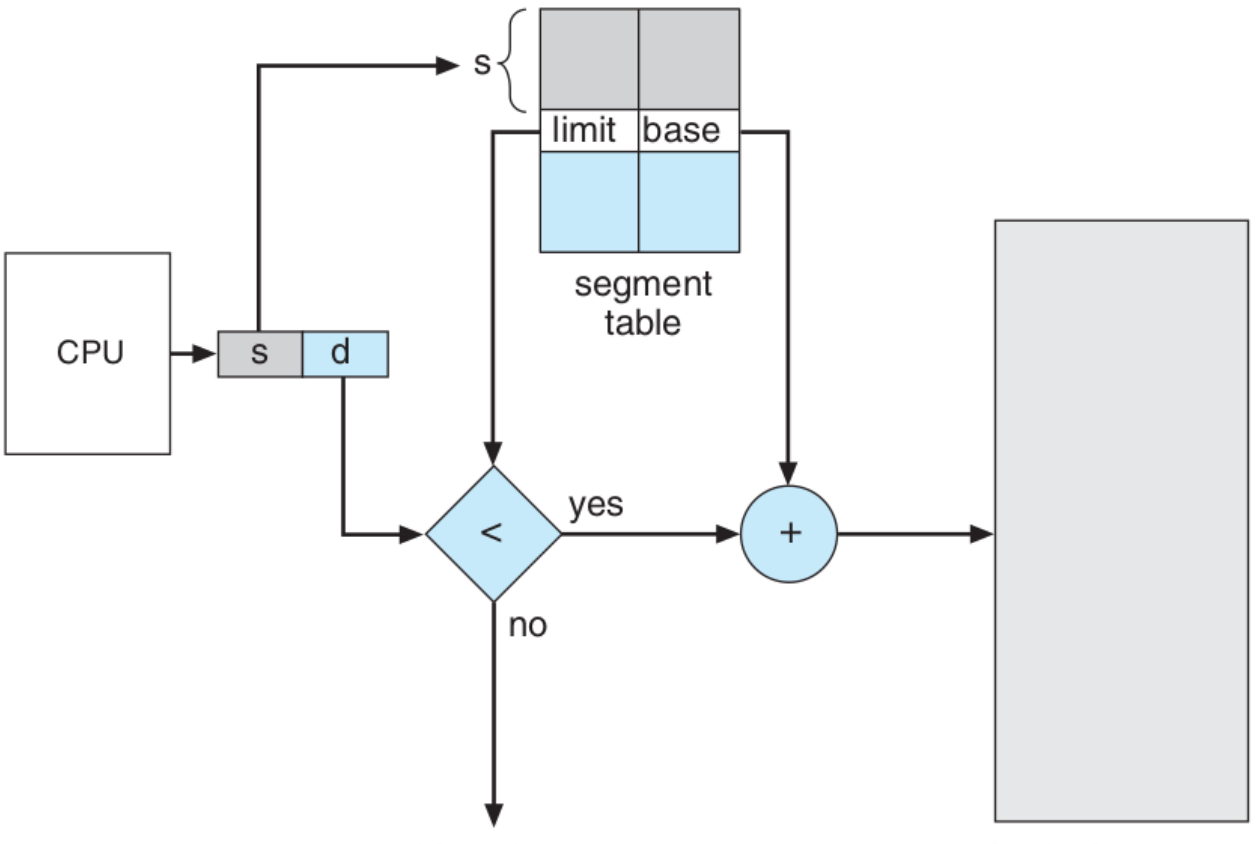


Segmentación

Es otro esquema de organización de la memoria, que se basa en dividir a un programa en secciones llamadas segmentos.



Los segmentos son unidades lógicas (programa principal, procedimientos, variables, etc), y se van cargando a memoria cuando se los necesita. Los segmentos pueden variar en cuanto al tamaño, por lo que puede darse una fragmentación externa.



trap: addressing error physical memory

El SO tiene una **tabla de segmentos**, que permite mapear la dirección lógica en física. Cada entrada tiene el Base Register (dirección física del comienzo del segmento) y el Limit Register (dirección física del final del segmento). Luego tenemos dos registros más, el *segment-table base register (STBR)* que apunta a la ubicación de la tabla de segmentos, y el *segment-table length register (STLR)* que contiene la cantidad de segmentos de un programa.

Segmentación Paginada

Como la paginación nos saca de encima el problema de la fragmentación externa, y la segmentación nos facilita la compartición y protección de la memoria, se mezclaron las dos en un nuevo esquema llamado segmentación paginada.

La segmentación paginada divide a un programa en segmentos, y cada segmento es dividido en páginas de tamaño fijo.

Algoritmos de Reemplazo

Óptimo

Selecciona a la página cuya próxima referencia es la más lejana a la actual. Este algoritmo es imposible de implementar ya que no se conocen futuros eventos. Es el que menos page faults genera (caso ideal).

FIFO

El más simple de implementar. Trata a los frames en uso como una cola circular, y la página más vieja en la memoria es reemplazada.

FIFO segunda chance

Se utiliza un bit adicional de referencia, que cuando la página se carga a memoria está en 0 y cambia cuando es referenciada. A la hora de buscar una página víctima, el algoritmo lo hará entre las que tengan el bit en 0, y mientras lo hace, va cambiando todas las que tienen el bit en 1 a 0.

Marco/Página	1	2	1	3	4	1	4	3	5
F1	1	1	1*	1*	1	1*	1*	1*	1
F2		2	2	2	4	4	4*	4*	4
F3				3	3	3	3	3*	5
PF?	X	X		X	X				X

$* = bit\ R = 1$

LRU (Least Recently Used)

Reemplaza la página que fue referenciada hace más tiempo. Para implementarlo es necesario que todas las páginas tengan un registro de cuando fué su última referencia.

Marco/Página	1	2	1	3	4	1	4	3	5
F1	1	1	1	1	1	1	1	1	5
F2		2	2	2	4	4	4	4	4
F3				3	3	3	3	3	3
PF?	X	X		X	X				X

Alcance de Reemplazo

Otro criterio a la hora de seleccionar páginas víctima es el alcance que puede llegar a tener el reemplazo.

- **Reemplazo global:** el fallo de página de un proceso puede reemplazar la página de cualquier otro proceso.
- **Reemplazo local:** el fallo de página de un proceso solo puede reemplazar sus propias páginas.