# Administración de memoria: Funciones y operaciones

Gunnar Wolf

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

# Índice



Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

# El administrador de memoria

- Es otra de las partes *fundamentales* de un sistema operativo
- En toda computadora basada en von Neumann, la memoria es el *único* almacenamiento a que tiene acceso directo el procesador
  - Todo otro almacenamiento tiene que pasar a través de controladores externos
  - Típicamente manejado a través de *memoria mapeada* o *de acceso directo*
- Todo proceso que ejecutemos debe estar *en* memoria
  - El administrador de memoria es el encargado de permitir que varios procesos la compartan

# Notación: Sistemas binario, decimal, hexadecimal

¡Importante!

- Para comprender bien este tema, así como el siguiente, es importante tener un buen dominio de conceptos básicos de los sistemas de numeración
  - Particularmente el binario, en segundo término el hexadecimal
- Aspectos principales:
  - Potencias de 2
  - Representación numérica en los diferentes sistemas
  - Máscaras de bits

Hay tarea al respecto.

# La memoria caché

- El manejo del caché es *casi* transparente para el sistema operativo
- Busca resolver el diferencial (creciente) de velocidad entre el CPU y la memoria
  - La memoria de alta velocidad que maneja el caché es mucho más cara
  - Comienza a entrar en juego la velocidad de los electrones sobre pistas de cobre
- Vimos ya que el OS tiene que saber al respecto: *Afinidad* de procesos a CPU
- Cuando se produce una *falla de caché* (no tiene copia de la dirección solicitada), es necesario *detener* al CPU
  - Insertando una *burbuja* o `NOOP` → *stall* (detención)

# ¿Por qué el caché?: Localidad de referencia

Se ha observado que prácticamente todos los procesos responden al principio de *localidad de referencia*:

**Localidad temporal** Es probable que un recurso que fue empleado recientemente vuelva a ser empleado en un futuro cercano.

**Localidad espacial** La probabilidad de que un recurso *aún no requerido* sea accesado es mucho mayor si fue requerido algún recurso cercano.

**Localidad secuencial** Un recurso, y muy particularmente la memoria, tiende a ser requerido de forma secuencial. (Caso especial de *Localidad espacial*)

# Ilustrando la localidad de referencia

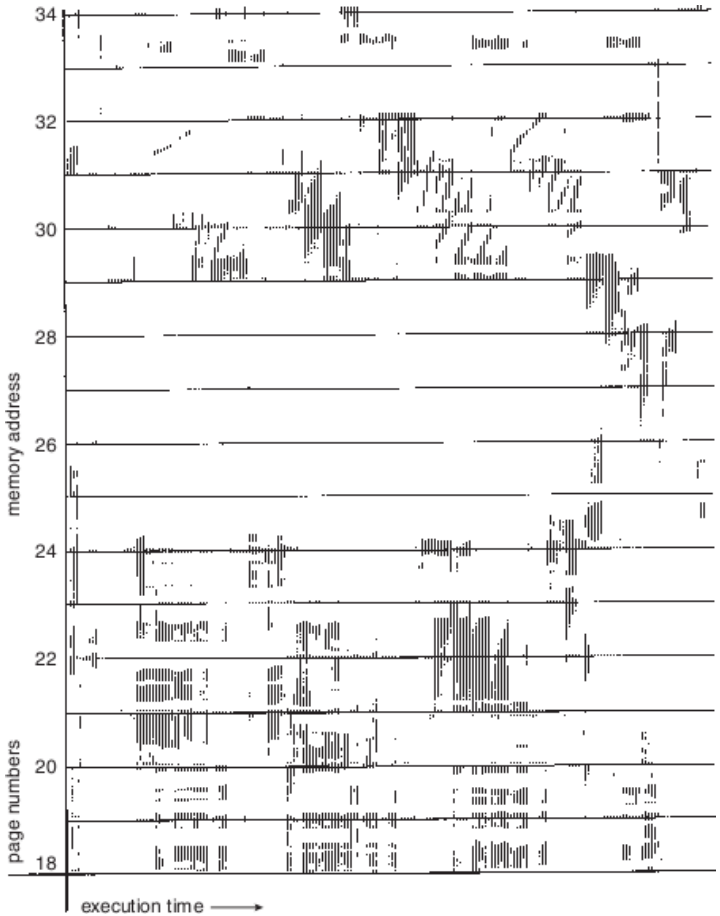



Figura: Patrones de acceso a memoria (Silberschatz, p.350)

# Índice



Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

# Abstracción de la memoria

- La memoria se presenta ante el CPU como un arreglo direccionable
- Cada byte tiene una dirección única y consecutiva
  - En algunas arquitecturas, las solicitudes deben estar *alineadas* (por ejemplo, a 64 bits), y no estarlo causa ua *trampa* o *falta*, incurriendo en demoras
  - Sin embargo, las direcciones siguen siendo de cada *byte*
- Para operar sobre *bits* específicos, tenemos que pedir el byte, trabajar sobre de él y enviarlo de vuelta — Como una unidad.


Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

# Espacio de direccionamiento: ¿Hasta dónde puedes contar?

- Cada arquitectura de procesador tiene un *espacio de direccionamiento* determinado
  - Siempre es una potencia de 2, y *casi siempre* en números *cerrados* (8, 16, 32, 64)
- Es el máximo de memoria que dicho procesador puede ver

  *Virtual* Lo máximo que la arquitectura ofrece como una dirección de memoria

  *Físico* Lo máximo que *un procesador específico* puede direccionar (limitado por su número de pines)


Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

# Procesadores de 16 bits

- Un procesador con *espacio de direccionamiento* de 16 bits puede referirse *directamente* a hasta $2^{16}$ (65,536) bytes.
- Principales procesadores de este tipo: Intel 8080 y 8085, Zilog Z80, MOS 6502 y 6510
- Intel 8086/8088: Direcciona hasta 20 bits (1024KB)
  - Pero al ser una arquitectura *real* de 16 bits, requiere emplear *segmentación* para alcanzar toda su memoria
- Intel 80286: espacio de direccionamiento de 24 bits (16 MB)
  - Pero su mercado era el mismo que el del 8086/8088, y por ciertas restricciones de su arquitectura, casi nunca se utilizó en estos modos

# Procesadores de 32 bits

- Hoy en día, los procesadores dominantes son de 32 o 64 bits
- Un procesador de 32 bits puede direccionar hasta 4GB — $2^{32}$ bytes (4,294,967,296 bytes)
  - Hoy ya está dentro del rango de lo *alcanzable*
  - Por medio de *PAE* (Extensión de Direcciones Físicas, *Physical Address Extension*), puede extenderse hasta $2^{52}$ (aunque típicamente $2^{36}$, 64GB)
  - PAE requiere un nivel adicional de *indirección* (lo veremos cuando cubramos *paginación*), por lo que pierde *un poco* de rendimiento, y cada proceso sigue viendo sólo hasta 4GB.

# Procesadores de 64 bits

- Un procesador de 64 bits puede direccionar hasta 16 exabytes — $2^{64}$ bytes (18,446,744,073,709,551,616 bytes)
- El hardware actual está limitado por un criterio económico a entre $2^{34}$ y $2^{48}$ bits, 16GB y 256TB
  - El costo de más de 256TB RAM hace extremadamente improbable que sea requerido en suficientes procesadores de propósito general
    - En un sistema *actual* de escritorio, es muy poco probable llegar a requerir incluso más de 16GB
  - Cada bit de direccionamiento necesario requiere un pin en el CPU, por tanto, tiene un costo económico directo

# ¿Más allá?

- ¿Qué tan lejos podemos llegar?
- El número total de átomos en el universo está estimado en cerca de $2^{80}$
- ¿Cuánto espacio (físico) necesitamos para nuestra memoria?
  - Por más que avance la miniaturización
- ¿Cuánto ancho de banda podemos esperar tener?
  - ¿Cómo podemos esperar llenar 64 bits de datos?


Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

# Índice



Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

# ¿Qué es / qué hace el MMU?

- Casi todos los sistemas operativos modernos *requieren* de una *unidad de manejo de memoria* (MMU)
  - Hardware, hoy es parte integral del CPU
- Trabaja *muy* de cerca con el sistema operativo
- Encargado de funciones de control de permisos, seguridad, y traducción de direcciones
- "Vigilando" todos los accesos a memoria que ejecuta el código
- Existen sistemas operativos multitarea que pueden funcionar sin MMU
  - Pagan como precio la confiabilidad del sistema

# ¿Traducción de direcciones?

- Es común que un sistema requiera más memoria *de la que está directamente disponible*
  - Más memoria de la que existe
  - Más memoria de la que el hardware puede direccionar
- Un proceso no tiene por qué conocer los detalles de la asignación de memoria — Le damos una vista virtual simplificada
  - ¿Cuántos bloques de memoria me asignaron?
  - ¿Cuál es la ubicación de cada uno de ellos?
  - ¿Qué pasa si intento escribir (o leer) de donde tengo prohibido?
- A lo largo de esta unidad iremos viendo las estrategias para responder a esto.

# Permisos

- El MMU verifica que un proceso no tenga acceso a datos de otro
  - A menos que sea *expresamente permitido* (vía el sistema operativo)
  - Hacer esta verificación desde el sistema operativo mismo incurriría en costos demasiado grandes
  - Una verificación estática no puede ser suficientemente exhaustiva
    - Direcciones construidas al vuelo
    - Modos indirectos de referencia a memoria
    - ...

# Protección de acceso: Primera aproximación

La arquitectura nos ofrece dos registros de uso *específico*, y su modificación requiere una operación privilegiada / *modo supervisor*:

**Registro base** Apunta a la primer dirección de memoria que pertenece a este proceso

**Registro límite** Cantidad de memoria que pertenece a este proceso a partir del registro base

*NOTA:* Esto asume que los bloques de memoria asignados a cada proceso sean *contiguos*

# Registros base y límite: Ejemplo (1)

- A un proceso le fue asignado un espacio de 64K (65535 bytes)
- A partir de la dirección 503808 (492K)

Esto se traduce a:

- El *registro base* contiene 503808 (0x7B000)
- El *registro limite* contiene 65535 ($2^{16}$, 0xFFFF)


Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

# Registros base y límite: Ejemplo (2)

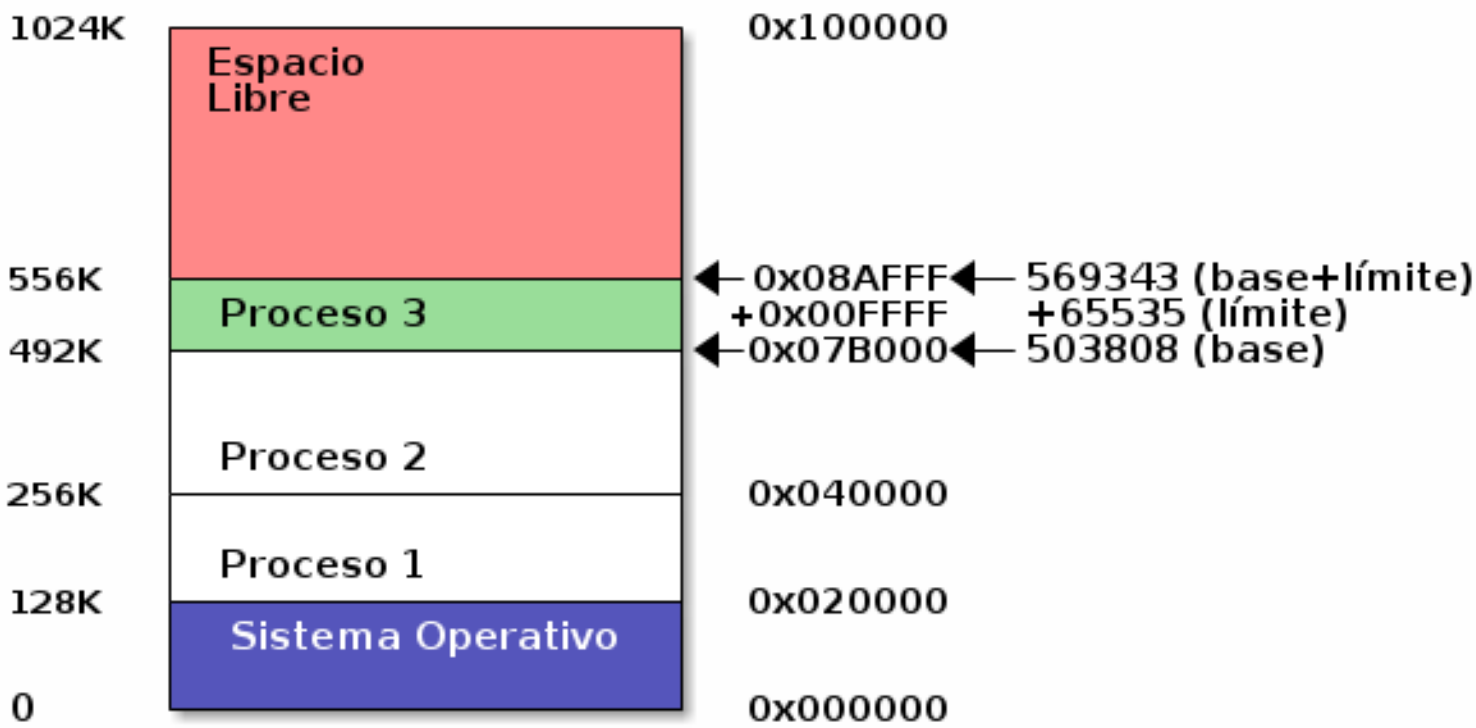



Figura: Espacio de direcciones válidas para el proceso 3 definido por un registro base y un registro límite

# Registros base y límite: Ejemplo (3)

Comportamiento del MMU

- Solicita acceso a *direcc* $< 503808 \rightarrow$ Falla:
  - Violación de segmento (*segmentation fault*)
- Solicita acceso a $503808 \leq$ *direcc* $\leq 569343 \rightarrow$ OK
  - Se otorga acceso a la dirección
- Solicita acceso a $569343 <$ *direcc* $\rightarrow$ Falla:
  - Violación de segmento (*segmentation fault*)

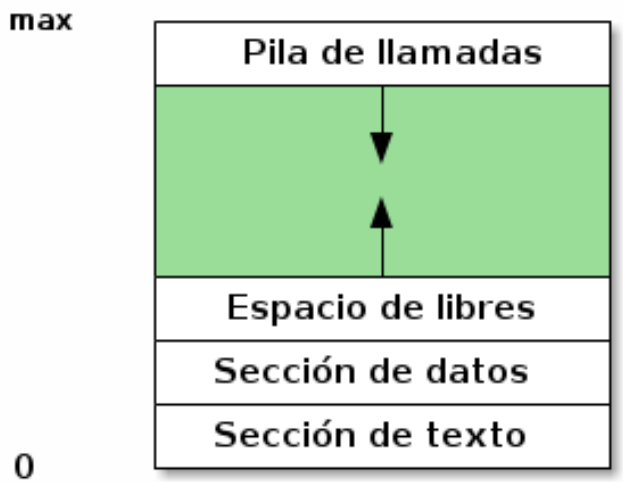
# Índice



Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

# Estructura de un proceso en memoria

- Para ejecutar un proceso, el SO *casi nunca* se limita a volcar el ejecutable a memoria
  - Excepto en sistemas antiguos: Formato `.COM` de MS-DOS, archivos `a.out` *reales* de Unixes antiguos
- Un proceso tiene diversas *regiones* de memoria, con usos muy diferentes:
  - Pila de llamadas
  - Espacio de *libres*
  - Sección de datos
  - Sección de texto



# Sección de texto

- Imagen en memoria de las instrucciones
- Las direcciones más *bajas* del espacio asignado
- El procesador va *avanzando* sobre este espacio mediante el registro de instrucción
- Su contenido *normalmente* no debería cambiar
  - Aunque hay código legal auto-modificable...
  - Afortunadamente, cada vez menos


Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

# Sección de datos

- Espacio fijo preasignado para variables *globales*
- Se fija en *tiempo de compilación*
- Su tamaño no puede cambiar
  - Aunque los *datos* sí cambien

Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

# Espacio de *libres* (*Heap*)

- Espacio empleado para la asignación dinámica de memoria
- Asignado *en tiempo de ejecución*
- En lenguajes que requieren *manejo manual* de la memoria, aquí se manejan todas las estructuras dinámicas
  - En C: `malloc`, `free`
  - En C++: `new`, `delete`
- En lenguajes con *gestión automática*, es monitoreado por los *recolectores de basura*
- *Crece hacia arriba*


Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

# Pila de llamadas (*Stack*)

- Estructuras representando la serie de funciones que han sido llamadas dentro del proceso, incluyendo:
  - Parámetros
  - Direcciones de *retorno*
  - Variables locales
  - etc.
- Ocupa la parte *más alta* del espacio en memoria
- *Crece hacia abajo*
- Indica *el punto actual de ejecución* del programa
  - Equivalente a decir *«aquí»*


Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

# Índice



Gunnar Wolf • sistop@gwolf.org • http://sistop.gwolf.org
Sistemas Operativos (1554) • Facultad de Ingeniería UNAM

# ¿Qué es la *resolución de direcciones*?

- Cuando escribimos un programa, sus funciones y variables son referidas por *nombre*
- El compilador va substituyendo los nombres por la *dirección en memoria* a donde debe referirse
  - Excepto en bibliotecas de ligado dinámico... Que abordaremos más adelante
- Pero en un sistema multiproceso, el compilador no necesariamente sabe dónde estará el espacio de memoria asignado al proceso
- Las direcciones indicadas en el *texto* del programa deben ser *traducidas* (o *resueltas*) a su ubicación definitiva
- Esto puede ocurrir en tres momentos...

# En tiempo de compilación

- El texto de programa tiene la dirección *absoluta* de datos y funciones
- Muy común en arquitecturas no multiprocesadas
- En las PC tempranas, el formato ejecutable `.COM` es un *volcado de memoria* con las direcciones absolutas
  - Formato limitado a *un segmento* de memoria (64K — 16 bits, ¿Recuerdan?)
- Hoy en día vemos esto en sistemas embebidos, microcontroladores, o de función específica
  - p.ej. Arduino
- P.ej. la variable `contador` queda traducida en la imagen *en disco* como su dirección: `510200`

# En tiempo de carga

- El sistema operativo tiene una rutina llamada *cargador* (*loader*)
  - Le asiste frecuentemente el *ligador* (*linker*), para incluir *en la sección de texto* todas las bibliotecas externas que requiera
- Analiza el texto del programa que va cargando, y actualiza a las referencias a memoria para apuntar al lugar correcto
  - Agregando el *desplazamiento* (*offset*) necesario (la dirección *base*)
- Depende de que el compilador indique la ubicación de cada una de las variables y funciones
- P.ej. `contador` queda traducida como un desplazamiento: `inicio + 5986`
  - Al cargarse, `inicio` es *resuelto* a 504214

# En tiempo de ejecución

- El programa *nunca* hace referencia a ubicaciones absolutas de memoria
  - El código generado por el compilador siempre indica *base* y *desplazamiento* (*offset*)
- Permite que el proceso sea reubicado en la memoria *incluso estando ya en ejecución*
- Requiere de apoyo en hardware (MMU)
- P.ej. la variable `contador` queda traducida como un desplazamiento: `inicio + 5986`
  - `inicio` se mantiene como etiqueta *durante la ejecución* y es resuelta cada vez que se requiera

# Proceso de compilación y carga

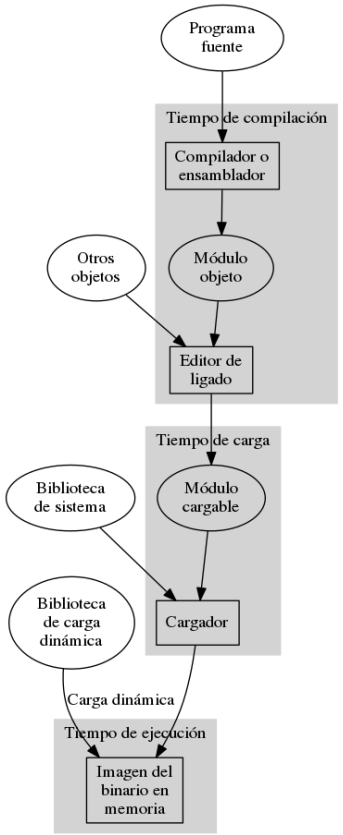



Figura: Proceso de compilación y carga de un programa, indicando el tipo de resolución de direcciones (Silberschatz, p.281)