Sistemas Operativos 1

Unidad 9 – Administración de Memoria

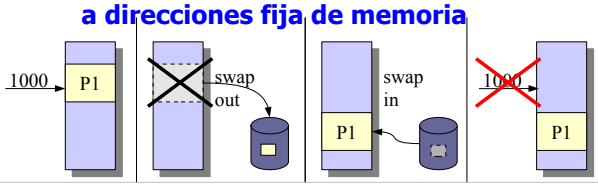
Prof. Dr. Alejandro Zunino
ISISTAN - CONICET

Introducción

- La administración de memoria se ocupa de:
  - llevar un registro de las partes de la memoria que están ocupadas y libres
  - asignar espacio a los procesos

1) Reubicación

- El Sistema Operativo puede reubicar procesos en memoria (swapping, compactación)
  - un proceso podría ocupar **diferentes lugares de memoria durante su tiempo de vida:**
- **Los programas no deberían estar atados a direcciones fija de memoria**



Administración de Memoria

- Objetivos
- Requerimientos
- Asignación contigua
- Introducción a Paginación

Objetivos y Requerimientos

- El objetivo es permitir que la mayor cantidad posible de procesos estén en memoria:
  - más procesos sugiere... mayor utilización de CPU
- Requerimientos:
  - 1) Reubicación
  - 2) Protección
  - 3) Memoria compartida
  - 4) Organización lógica
  - 5) Organización física

2) Protección

- Los procesos no deberían ser capaces de acceder a áreas de memoria pertenecientes a otros procesos sin permiso:
  - **No** es posible verificar las direcciones que los programas generan en **tiempo de compilación o carga debido** a que podría haber **reubicación**.
  - **La verificación debe realizarse durante la ejecución** (x hardware)

3) Memoria Compartida

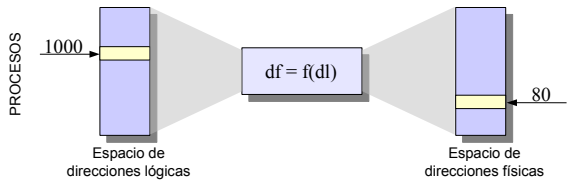
- Permitir a varios procesos acceder a una porción de memoria común sin sacrificar la protección:
  - Ej: para compartir una biblioteca con funciones comunes a varios procesos.
  - Ej: para compartir datos

5) Organización Física

- La memoria externa es el medio donde se perduran los programas y datos, mientras que la memoria principal alberga a los programas y datos en uso:
  - El movimiento de información entre esos dos niveles de memoria es una de las responsabilidades de la administración de memoria.
  - Podría ser ineficiente dejar esto en manos del programador

Direcciones Lógicas y Físicas

- **Dirección física:** es un lugar físico en la memoria
- **Dirección lógica**: referencia un lugar de memoria que es independiente de su organización

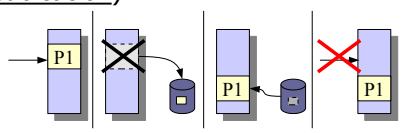


4) Organización Lógica

- La memoria es un arreglo de bytes
- Sin embargo, los programadores desarrollan módulos con diferentes características:
  - módulos de código: sólo lectura
  - módulos de datos: sólo lectura o lectura/escritura
  - módulos privados
  - módulos públicos
- El SO/hardware deberían proveer soporte para módulos

Direcciones Lógicas y Físicas

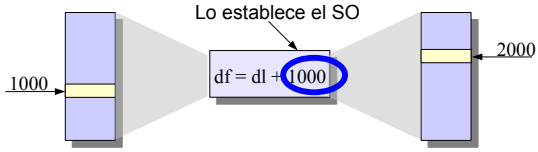
- Las direcciones de memoria generadas por un proceso no pueden ser siempre fijas (por la reubicación)



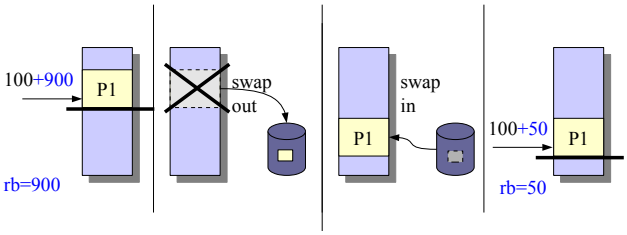
- Se resuelve ocultando la verdadera ubicación de los procesos en memoria.

Ejemplo: Reubicación con Registro Base

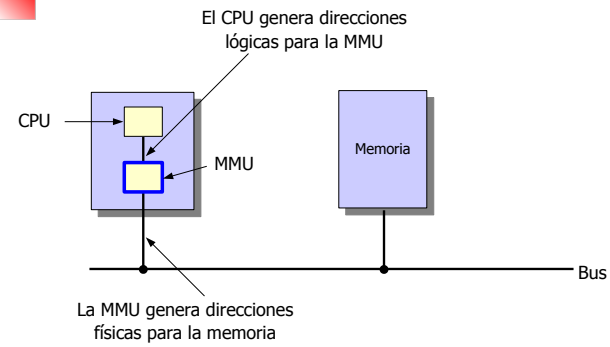
- Las direcciones físicas se calculan "al vuelo" conforme se ejecutan los procesos.
- Para que la performance sea adecuada la traducción debe ser hecha por hardware



Ejemplo: Reubicación con Registro Base



CPU, MMU y Memoria

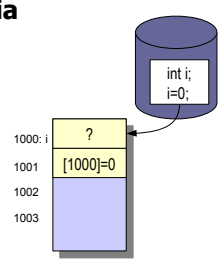


Cuando se Carga un Programa...

... deben ligarse sus instrucciones y datos a direcciones de memoria

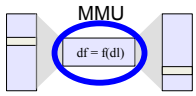
Cuándo?

- En tiempo de compilación
- En tiempo de carga
- En tiempo de ejecución

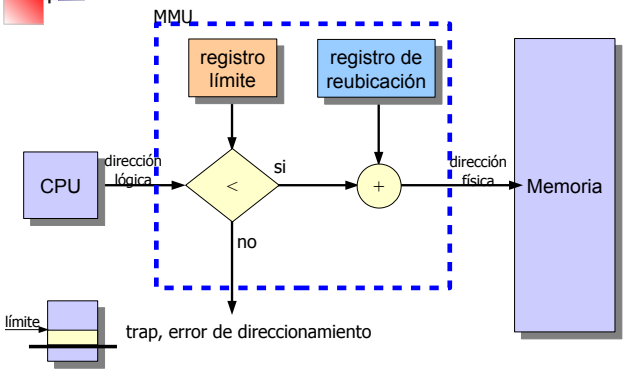


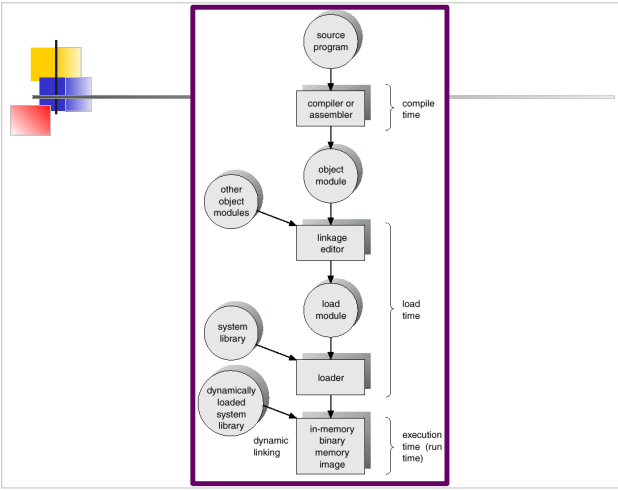
Unidad de Administración de Memoria (MMU)

- Dispositivo hardware que traduce direcciones lógicas a físicas
- Los procesos usan direcciones lógicas
  - **Nunca ven direcciones físicas**
  - **Nunca ven la organización física de la memoria**



Ejemplo: Registro de Reubicación y Registro Límite (Protección)





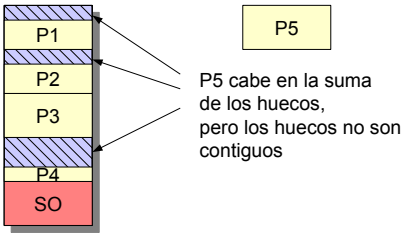
...se Ligan sus Instrucciones y Datos a Direcciones de Memoria...

- En tiempo de compilación
  - si la dirección de memoria inicial se conoce a priori... (sin MMU)
  - se debe recompilar si cambia la dirección...
- En tiempo de carga:
  - código relativo (sin MMU)
  - durante la carga se resuelven las direcciones relativas sumándoles un valor (cargador)... puede ser lento
- En tiempo de ejecución
  - se retrasa hasta la ejecución para permitir que los procesos puedan reubicarse
  - **requiere de hardware (MMU) (+ $$$)**

Asignación Contigua con Particiones Estáticas y Dinámicas

Asignación Contigua

- A veces parece haber espacio...
  - **Fragmentación externa**

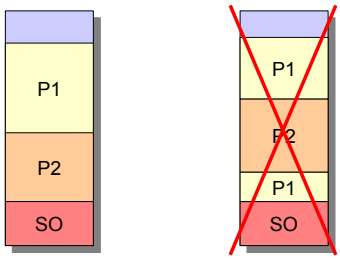


Cuando se Carga un Programa...

- Direcciones lógicas == Direcciones físicas
  - Ligado en tiempo de compilación y Ligado en tiempo de carga:
    - **No soportan reubicación**
- Direcciones lógicas != Direcciones físicas
  - Ligado en tiempo de ejecución
    - Soporta reubicación

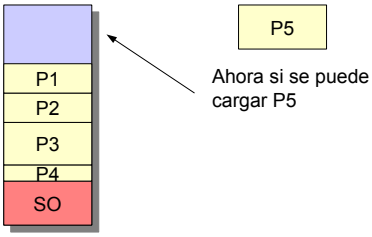
Asignación Contigua

- Un proceso debe ser cargado por completo en un área contigua de la memoria principal



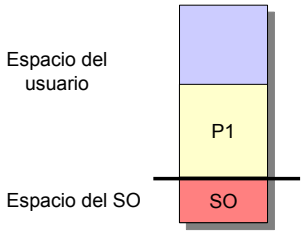
Asignación Contigua

- A compactar...
  - ... es muy costoso en tiempo
  - ... requiere de soporte para reubicación
  - ... suspende la ejecución de los procesos



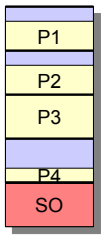
Particiones

- Usualmente la memoria se divide en dos o más particiones:



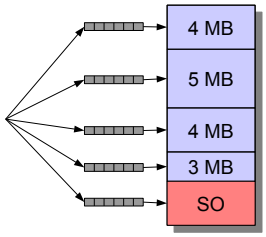
Asignación de Espacio: Particiones del Mismo Tamaño

- Si hay una partición disponible el proceso se carga
- Si no, se elige un proceso para swappear



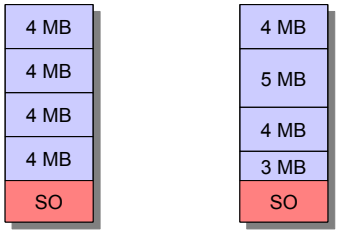
Asignación de Espacio: Particiones de Distinto Tamaño

- Best fit con múltiples colas:
  - asigna los procesos a la menor partición que lo contenga
  - intenta minimizar la fragmentación interna
  - bloques muy requeridos/poco requeridos
    - baja el grado de multiprogramación



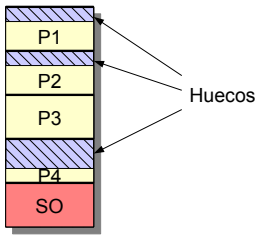
Particiones Estáticas

- Las particiones no se solapan entre sí.
- Pueden ser de distintos tamaños o iguales
- No cambian una vez que el sistema arranca



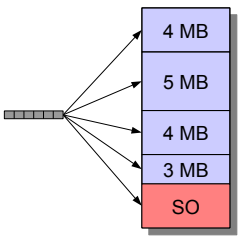
Asignación de Espacio: Particiones del Mismo Tamaño

- Los huecos que quedan libres se desperdician (1 proceso por partición)
  - **Fragmentación interna**



Asignación de Espacio: Particiones de Distinto Tamaño

- Best fit con una cola:
  - asigna los procesos a la menor partición que lo contenga
  - mayor multiprogramación
  - mayor fragmentación interna



Particiones Fijas: Conclusiones

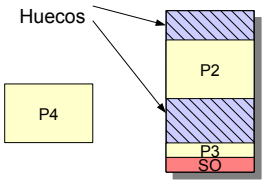
- Baja utilización de memoria:
  - Una partición por proceso
  - Fragmentación interna
- Qué pasa si un proceso crece?

Asignación de Espacio: Particiones Variables

- First-fit:
  - asigna el primer hueco con espacio suficiente
- Next-fit:
  - misma lógica que el anterior, pero busca desde el último hueco asignado
- Best-fit:
  - asigna el menor de los huecos con espacio suficiente
  - debe recorrer toda la lista
- Worst-fit:
  - asigna el mayor de los huecos con espacio suficiente
  - debe recorrer toda la lista

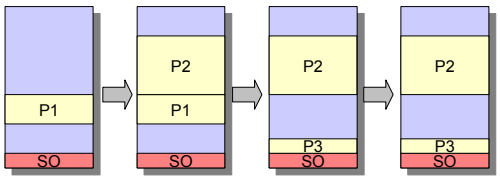
Particiones Variables: Conclusiones

- Baja utilización de memoria:
  - Fragmentación externa
    - Compactación...
  - Qué pasa si un proceso crece?



Particiones Variables

- Cuando se crea un proceso se le asigna un área libre de memoria con el tamaño suficiente para contenerlo
  - el área asignada es la partición
  - el tamaño y número de particiones es variable



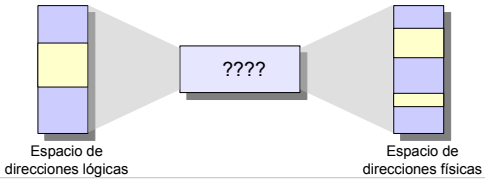
Asignación de Espacio: Particiones Variables

- First-fit tiende a asignar memoria en la parte baja (o alta):
  - tiende a crear menos fragmentación que Next-fit
- Best-fit tiende a generar muchos huecos pequeños:
  - mucha fragmentación
- First-fit y next-fit son generalmente utilizados

Introducción a Paginación

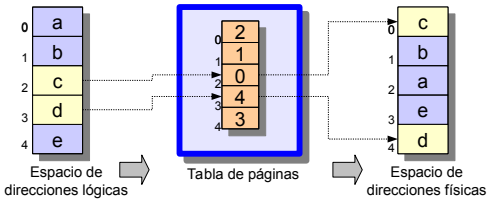
Paginación

- Asignación contigua tiene problemas:
  - fragmentación externa
  - crecimiento de procesos
- Idea... **ocultar a los procesos la organización de la memoria**



Traducción de Direcciones

- Tabla de páginas
  - convierte página a marcos

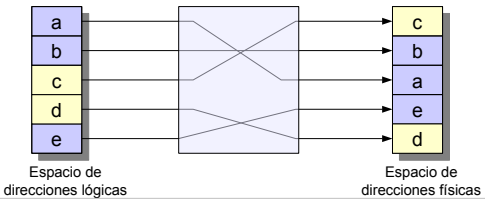


Paginación: Conclusiones

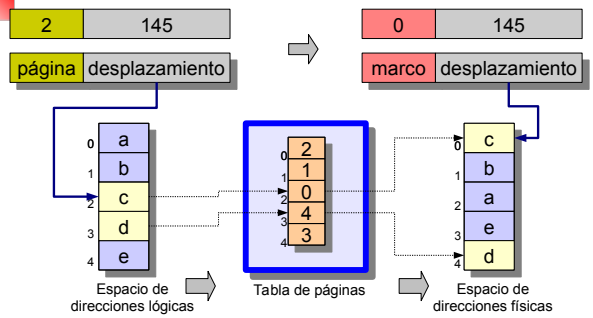
- El SO gana mayor libertad para organizar la memoria:
  - no es necesario que la asignación sea contigua
- Desaparece la fragmentación externa
- Aún hay fragmentación interna
  - ej: de una página de 4 KB sólo se usan 3 KB
- La tabla ocupa memoria...
- Velocidad???

Implementación

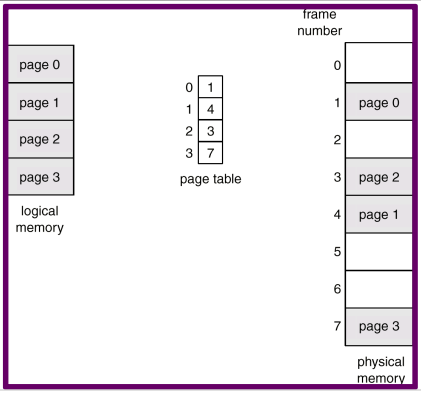
- Dividir la memoria en física en bloques de tamaño fijo llamados marcos (frames) de 1KB, 2KB, 4KB...
- Dividir la memoria lógica en bloques del mismo tamaño llamados páginas
- Traducir entre páginas y marcos



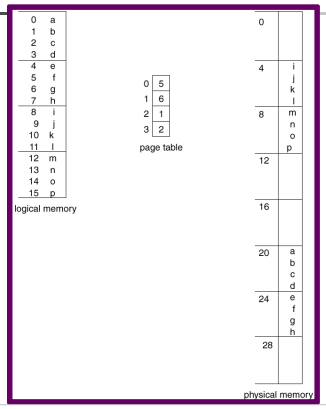
Traducción de Direcciones



Paging Example



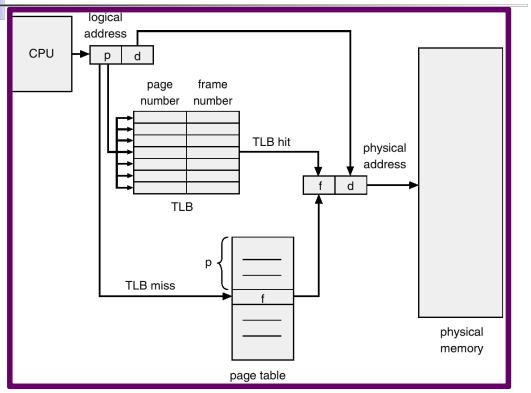
Paging Example



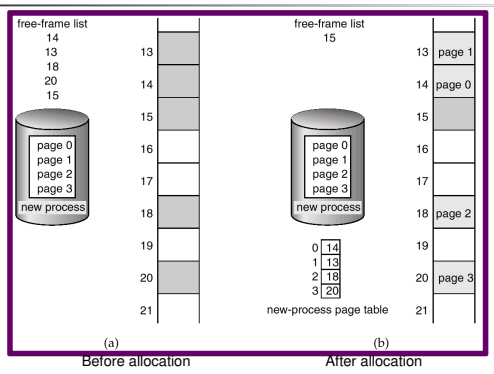
Implementation of Page Table

- Page table is kept in main memory
- **Page-table base register (PTBR)** points to the page table
- **Page-table length register (PRLR)** indicates its size
- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**
- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process

Paging Hardware With TLB



Free Frames



Associative Memory

- Associative memory – parallel search

Page #	Frame #

- Address translation (A´, A´´)
  - If A´ is in associative register, get frame # out.
  - Otherwise get frame # from page table in memory

Effective Access Time

- Associative Lookup = ε time unit
- Assume memory cycle time is 1 microsecond
- Hit ratio – percentage of times that a page number is found in the associative registers; ration related to number of associative registers.
- Hit ratio = α
- Effective Access Time (EAT)

$$EAT = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$$
$$= 2 + \varepsilon - \alpha$$

Memory Protection

- Memory protection implemented by associating protection bit with each frame.
- *Valid-invalid* bit attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page.
  - "invalid" indicates that the page is not in the process' logical address space.

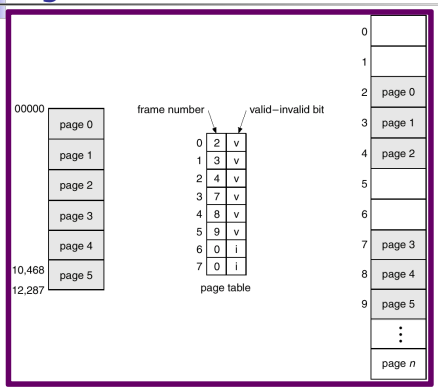
Shared Pages

- Shared code
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in same location in the logical address space of all processes.
- Private code and data
  - Each process keeps a separate copy of the code and data.
  - The pages for the private code and data can appear anywhere in the logical address space.

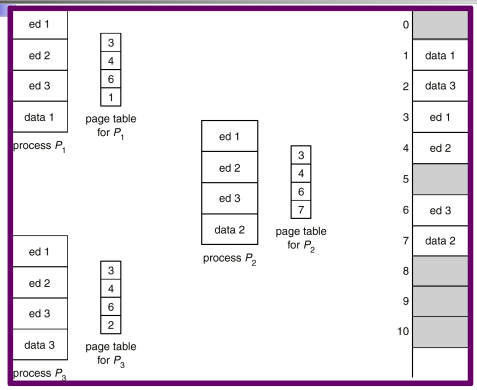
Page tables can get HUGE

- Consider a 32-bit logical address space as on modern computers
  - Page size of 4 KB (2^{12})
  - Page table would have 1 million entries (2^{32} / 2^{12})
  - If each entry is 4 bytes -> 4 MB of physical address space / memory for page table alone
  - That amount of memory used to cost a lot
  - Don't want to allocate that contiguously in main memory

Valid (v) or Invalid (i) Bit In A Page Table



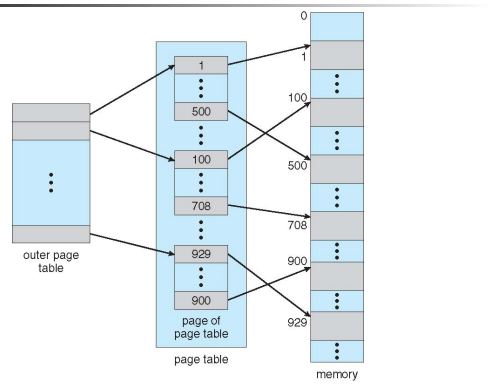
Shared Pages Example



Hierarchical Page Tables

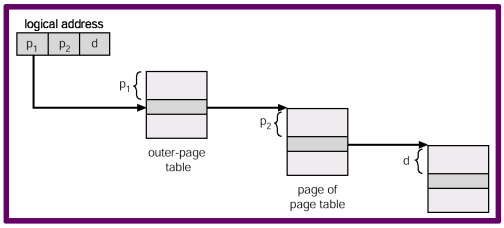
- Break up the logical address space into multiple page tables.
- A simple technique is a two-level page table.

Two-Level Page-Table Scheme



Address-Translation Scheme

- Address-translation scheme for a two-level 32-bit paging architecture



64-bit Logical Address Space

- But in the following example the 2nd outer page table is still 2^{34} bytes in size
- And possibly 4 memory access to get to one physical memory location

outer page	inner page	offset
p_1	p_2	d
42	10	12

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits.
  - a page offset consisting of 10 bits.
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number.
  - a 10-bit page offset.
- Thus, a logical address is as follows:

page number		page offset
p_1	p_2	d
12	10	10

- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the outer table.

64-bit Logical Address Space

- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would look like

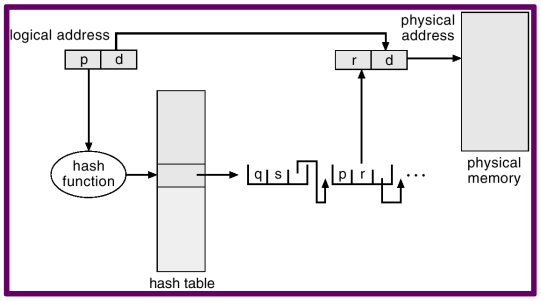
outer page	inner page	page offset
p_1	p_2	d
42	10	12

  - Outer page table has 2^{42} entries or 16384 GB
  - **One solution is to add a 2nd outer page table**

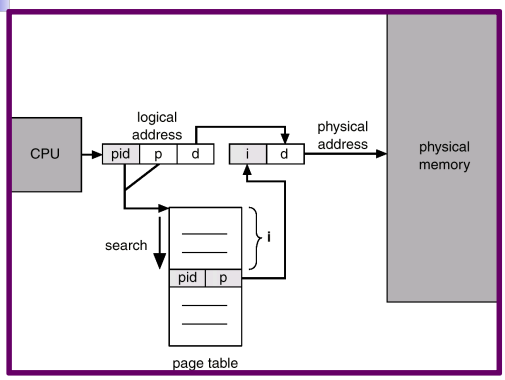
Hashed Page Tables

- Common in address spaces > 32 bits.
- The virtual page number is hashed into a page table. This page table contains a chain of elements hashing to the same location.
- Virtual page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted.

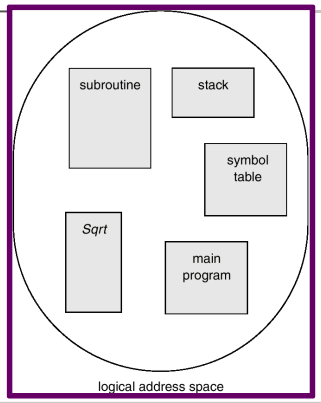
Hashed Page Table



Inverted Page Table Architecture



User's View of a Program



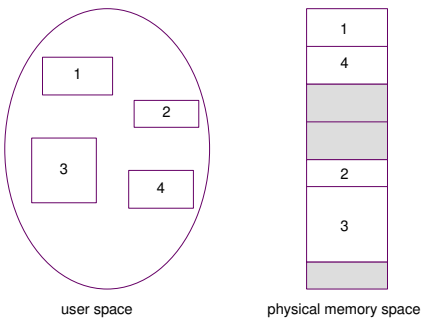
Inverted Page Table

- One entry for each real page of memory.
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs.
- Use hash table to limit the search to one — or at most a few — page-table entries.

Segmentation

- Memory-management scheme that supports user view of memory.
- A program is a collection of segments. A segment is a logical unit such as:
  - main program,
  - procedure,
  - function,
  - method,
  - object,
  - local variables, global variables,
  - common block,
  - stack,
  - symbol table, arrays

Logical View of Segmentation



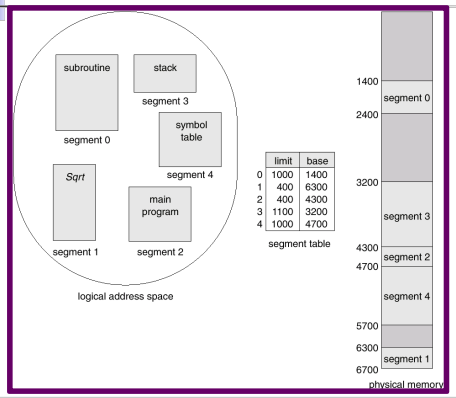
Segmentation Architecture

- Logical address consists of a two tuple:
  <segment-number, offset>,
- *Segment table* – maps two-dimensional physical addresses; each table entry has:
  - base – contains the starting physical address where the segments reside in memory.
  - *limit* – specifies the length of the segment.
- *Segment-table base register (STBR)* points to the segment table's location in memory.
- *Segment-table length register (STLR)* indicates number of segments used by a program;
- segment number *s* is legal if *s* < STLR.

Segmentation Architecture (Cont.)

- Protection. With each entry in segment table associate:
  - validation bit = 0 ⇒ illegal segment
  - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level.
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem.
- A segmentation example is shown in the following diagram

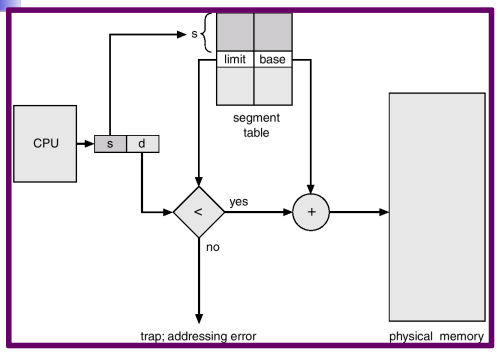
Example of Segmentation



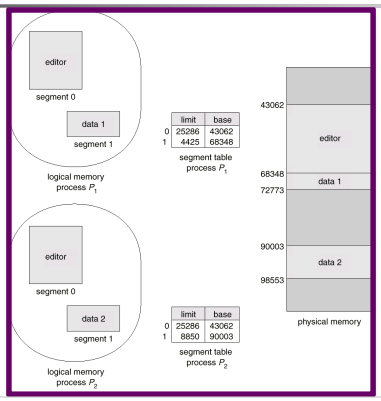
Segmentation Architecture (Cont.)

- Relocation.
  - dynamic
  - by segment table
- Sharing.
  - shared segments
  - same segment number
- Allocation.
  - first fit/best fit
  - external fragmentation

Segmentation Hardware



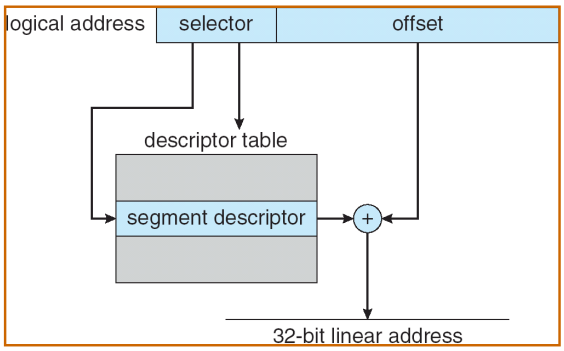
Sharing of Segments



Example: The Intel Pentium

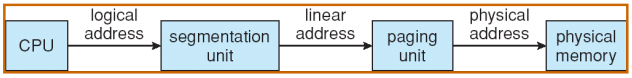
- Supports both segmentation and segmentation with paging
- CPU generates logical address
  - Given to segmentation unit
    - Which produces linear addresses
  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU

Intel Pentium Segmentation



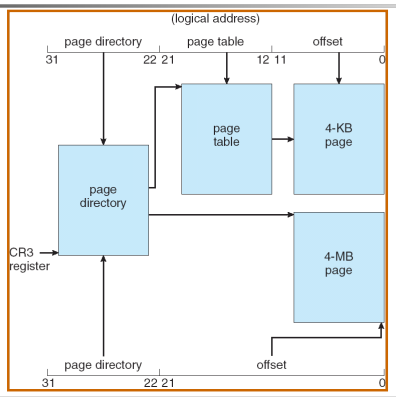
Linear Address in Linux

Logical to Physical Address Translation in Pentium



page number		page offset
p_1	p_2	d
10	10	12

Pentium Paging Architecture



Three-level Paging in Linux