Operating Systems

Main Memory

Giorgio Grisetti

`grisetti@diag.uniroma1.it`

Department of Computer Control and Management Engineering
Sapienza University of Rome

Facts

Main memory and registers are the only storage that CPU can access directly

Memory unit only sees a stream of addresses + read requests, or address + data and write requests

Register access in one CPU clock (or less)

Main memory can take many cycles, causing a **stall**

Cache sits between main memory and CPU registers

Protection of memory required to ensure correct operation

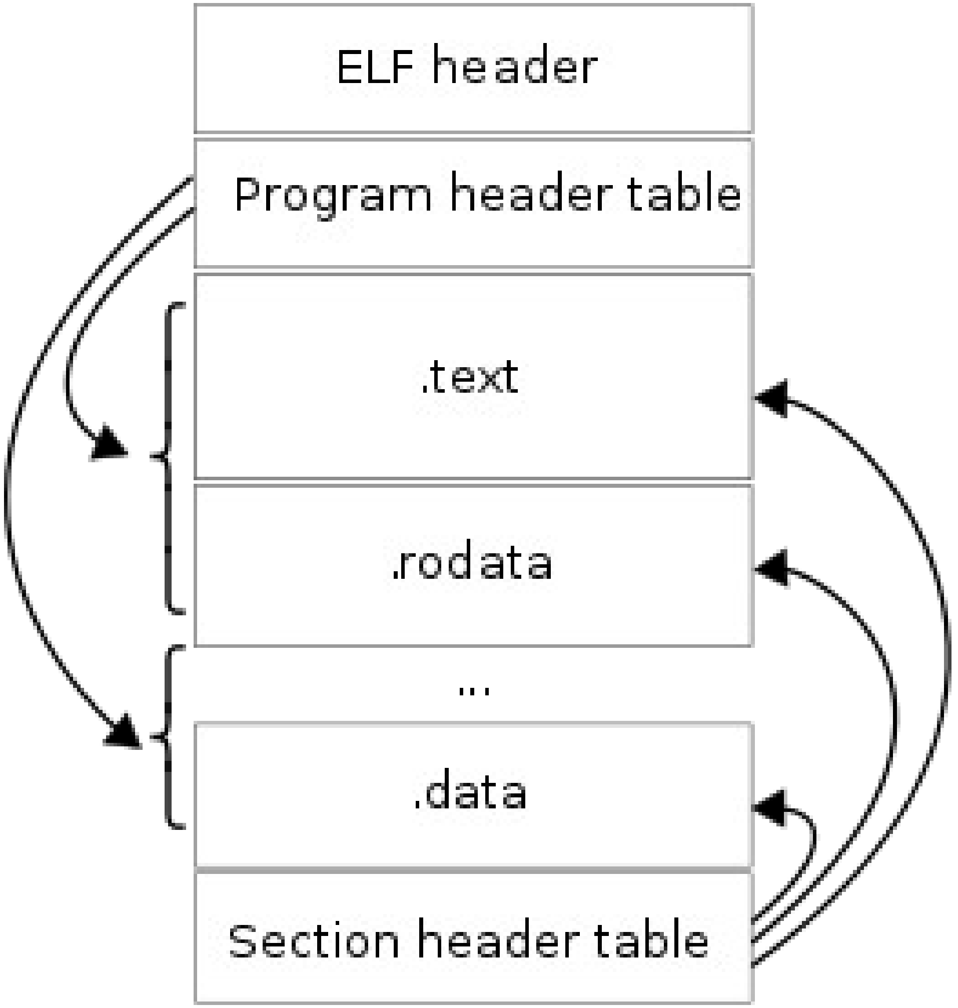
Loading a Program

When starting a program file from disk with exec, several actions take place

- the program file header is decoded to extract information about
  - memory segments (.text, .bss, .data)
  - dynamic libraries requested
- the dynamic linker:
  - loads the dynamic libraries from the corresponding program files
  - decodes the them and maps the libraries in the program space
  - adjust the dangling references to symbols defined in the libraries and declared/used in the program
  - this procedure might be recursive
- The address used by the program might need to be adjusted to match the actual memory layout

ELF header

- The executable programs are stored on the disk as binary files containing all information needed to load a program image in memory
  - Program code
  - Initialized Variables
  - Dynamic linking information
  - ....



Address Binding

Programs on disk, ready to be brought into memory to execute form an **input queue**

- Without HW support, must be loaded into address 0000

Inconvenient to have first user process physical address always at 0000

- How can it not be?

Further, addresses represented in different ways at different stages of a program’s life

- Source code addresses usually symbolic
- Compiled code addresses **bind** to relocatable addresses
  - i.e. “14 bytes from beginning of this module”
- Linker or loader will bind relocatable addresses to absolute addresses
  - i.e. 74014
- Each binding maps one address space to another

Address Binding

Address binding of instructions and data to memory addresses can happen at three different stages

- **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
- **Load time**: Must generate **relocatable code** if memory location is not known at compile time
- **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another
    - Needs hardware support for address mapping

Dynamic Linking

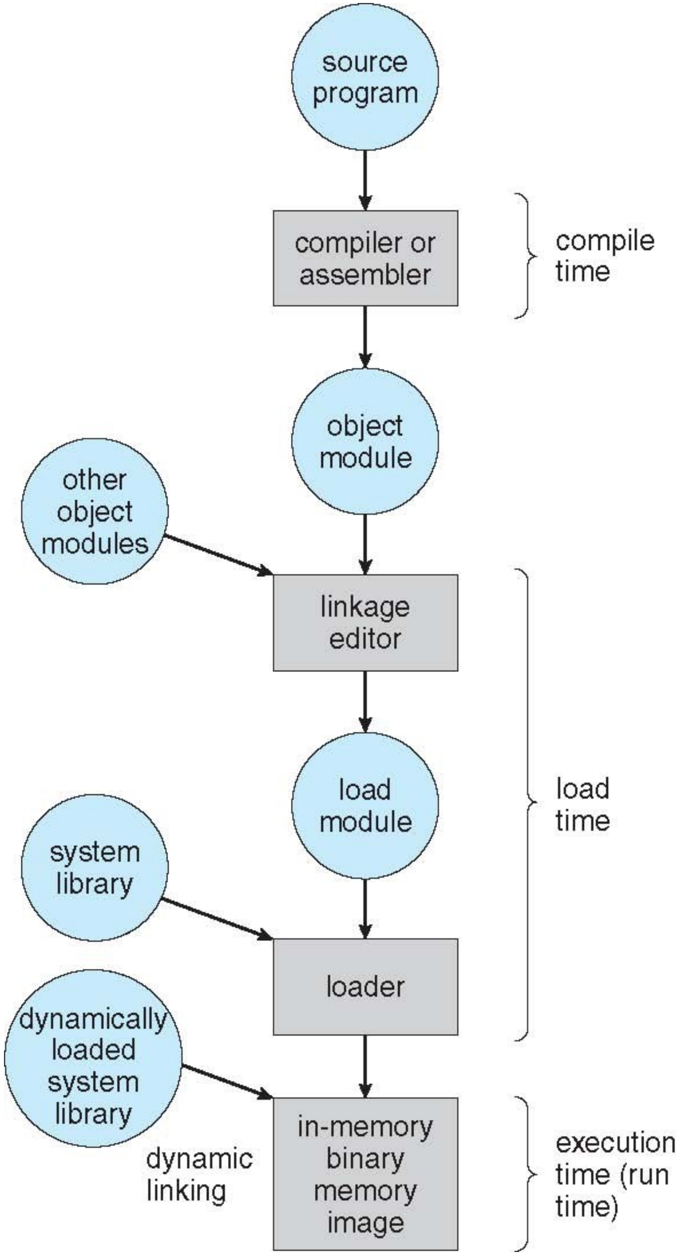
Static linking – system libraries and program code combined by the loader into the binary program image

Dynamic linking –linking postponed until execution time

- Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system checks if routine is in processes' memory address
  - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries
- Known as **shared libraries**
- Consider applicability to patching system libraries
  - Versioning may be needed

Phases of a Program

- source: addresses are symbols
- object files: addresses are labels
- load time: labels are filled with real addresses
- execution time: labels are filled with address populated by loading modules



Allocation

Partition the memory to give each process what it needs

Issues:

- the memory need can change during the life of the process:
  - module can be loaded/unloaded
  - heap size can vary
- fragmentation: there could be enough memory, but no right "hole" to satisfy a process.
- protection: prevent a user process from
  - reading/writing on memory not owned by it
  - executing code in "non executable" areas

Allocation

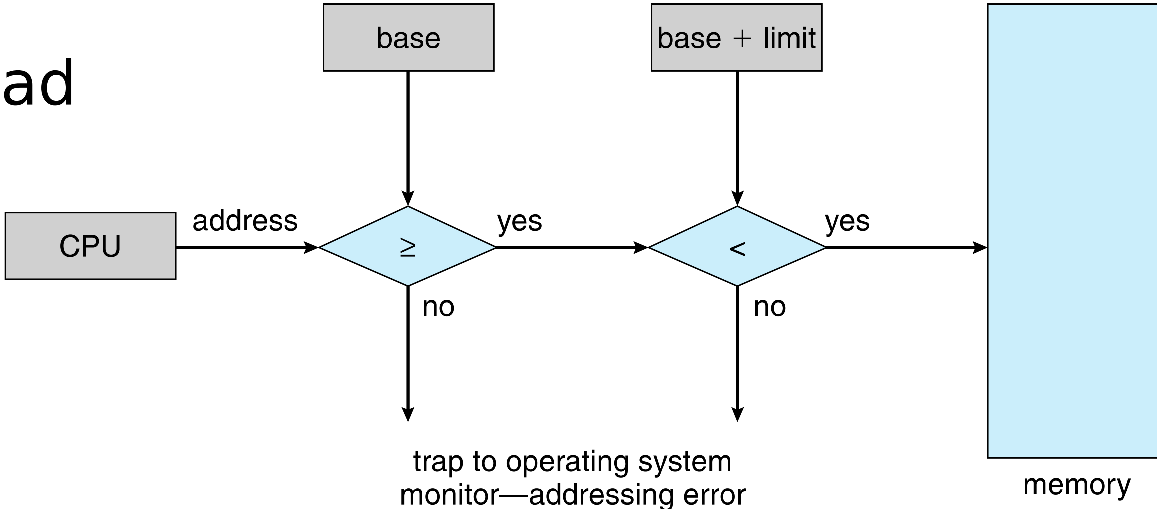
In the next slides an "historical" view of hardware technique used to approach the various problems.

Nowadays paging is mostly used, rest obsolete.

Protection

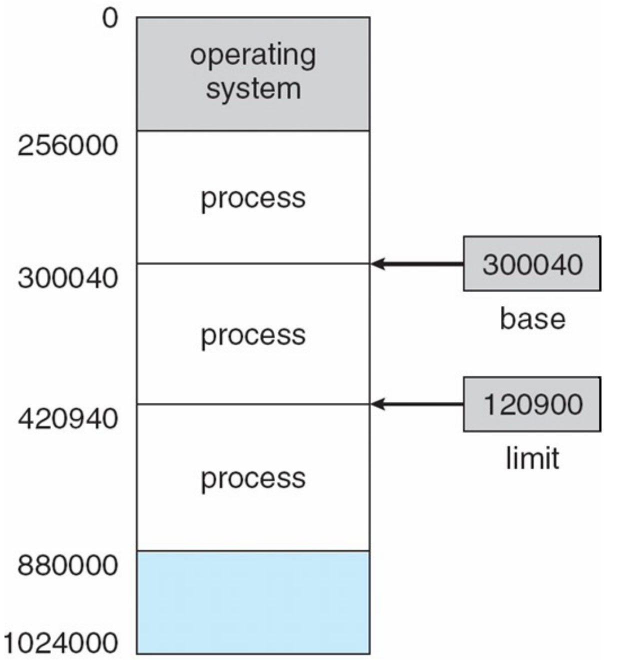
How to ensure that a program does not write/read in the address space of another?

How to ensure that a program does not access kernel memory?



Early solution: Base and Limit Registers

- Each time the CPU generates an address, some logic checks that its value is within the bounds of base and limit
- If this is not the case -> exception

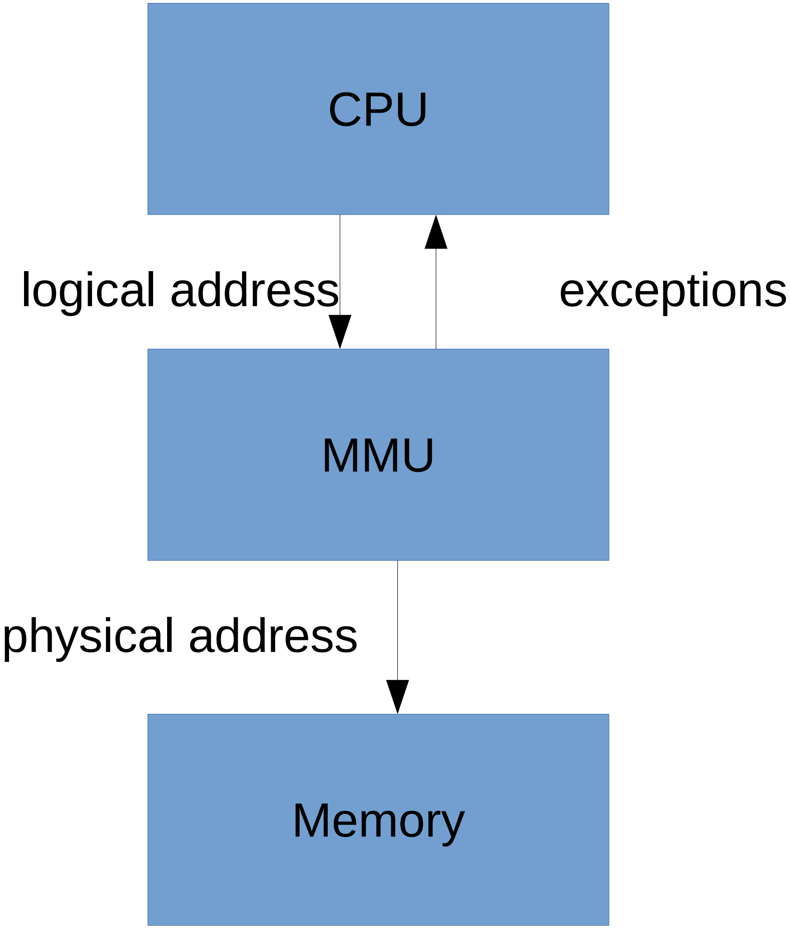


Logical and Physical Addresses

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the RAM chips (on the address bus)
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program

Memory Management Unit

- Hardware device that maps virtual to physical address at run time
- CPU is UNAWARE of physical addresses, it can only "configure" the MMU
- Different configurations of the MMU produce different physical addresses from the same logical address



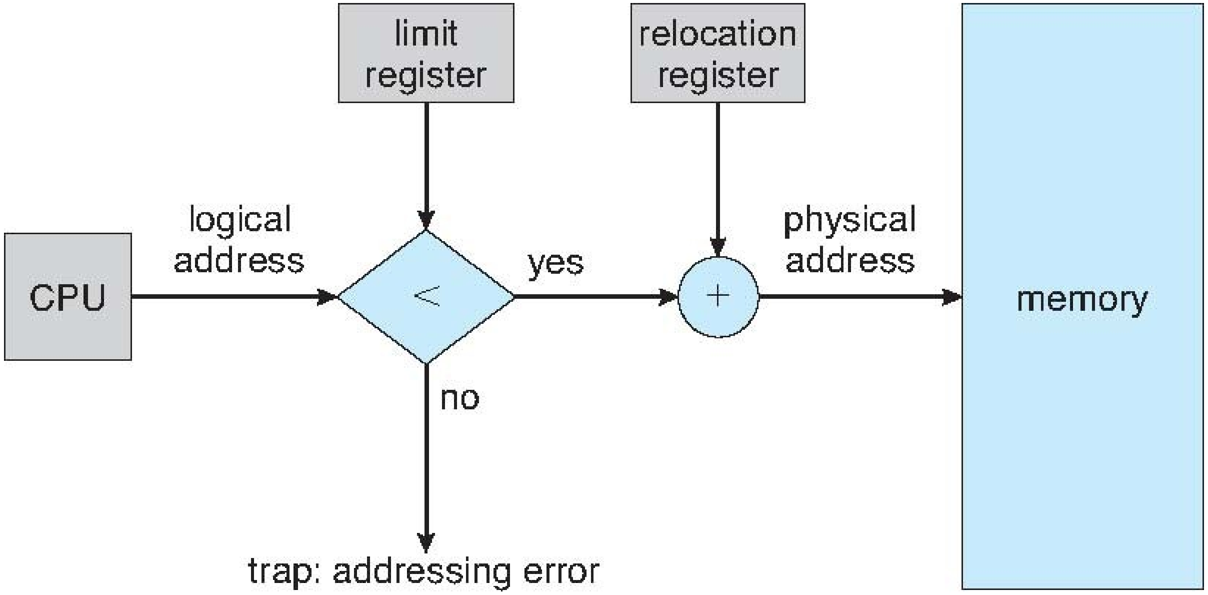
Relocation and Limit (HW)

Solves Protection:

- A pair of **relocation** and **limit registers** define the logical address space

  physical address=relocation+logical address

- CPU must check every memory access generated in user mode to be sure it is below the limit register



Contiguous Allocation

Relocation/limit registers used to protect user processes from each other, and from changing operating-system code and data

- Base register contains value of smallest physical address **for the process**
- Limit register contains range of logical addresses – each logical address must be less than the limit register
- MMU maps logical address *dynamically*
- Can then allow actions such as kernel code being **transient** and kernel changing size

Multiple Partition Allocation

Give each process a contiguous range of addresses

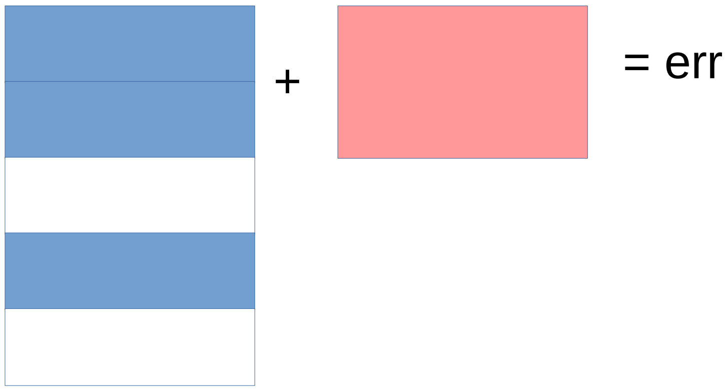
When a program terminates its memory is freed

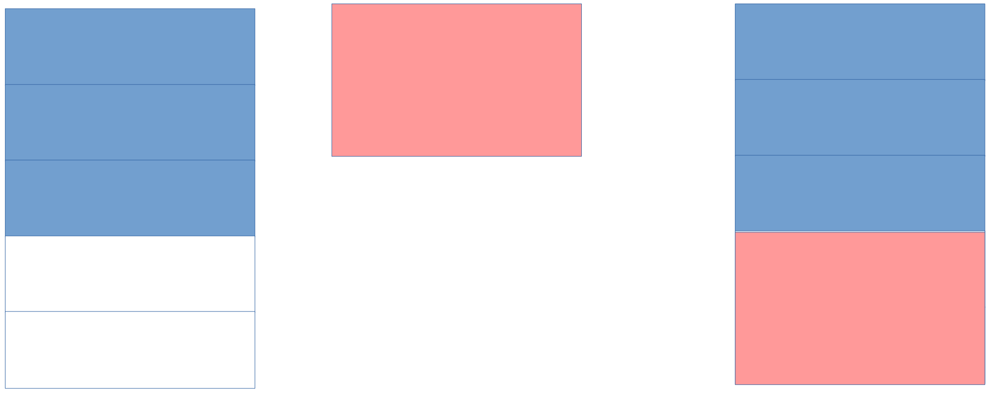
Fixed size: number of programs limited by memory

- Issues:
  - limits # of processes, wasted memory
  - internal fragmentation (unused memory in partition)

Variable size: amount of program memory can vary

- Issues:
  - external fragmentation: there might be enough free memory but not contiguous to accommodate a new process





Multiple Partition Allocation

How to assign a "bucket" to a process?

- **First-fit**: Allocate the ***first*** hole that is big enough

- **Best-fit**: Allocate the ***smallest*** hole that is big enough

- **Worst-fit**: Allocate the ***largest*** hole

First-fit and best-fit better than worst-fit in terms of speed and storage utilization

Compaction

Reduce external fragmentation by **compaction**

- Shuffle memory contents to place all free memory together in one large block
- Compaction is possible *only* if relocation is dynamic, and is done at execution time
- I/O problem
  - Latch job in memory while it is involved in I/O
  - Do I/O only into OS buffers

- Now consider that backing store has same fragmentation problems

Segmentation

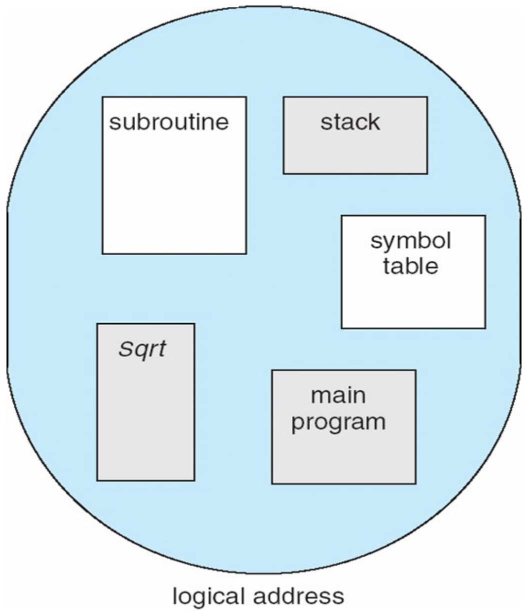
Earliest processors had address spaces larger than their registers

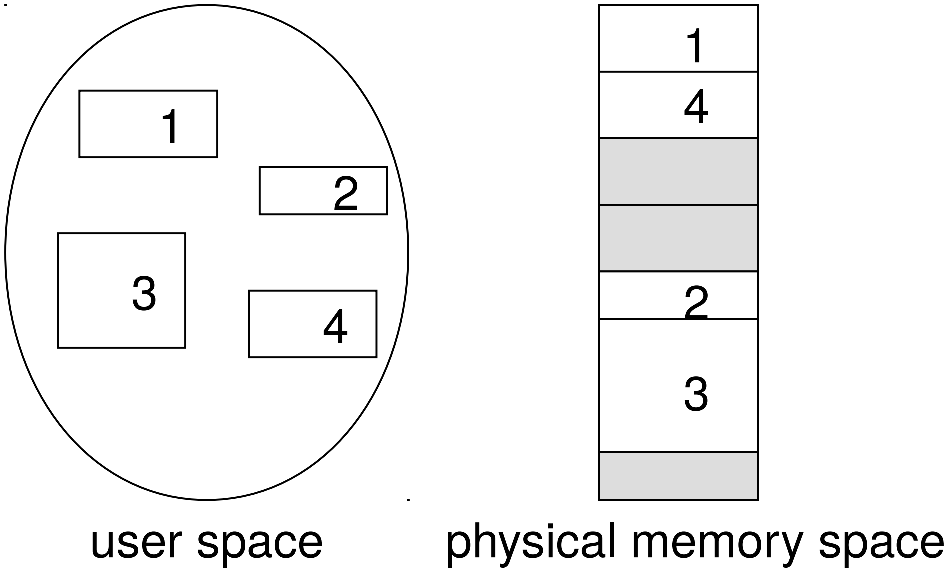
How to access larger amount of RAM?

- Use an external register that stores the higher bits of the address

Segmentation (HW)

- Earliest processors had address spaces larger than their registers
- how to access larger amount of RAM?
  - use of segment registers, that store the higher bits of an address (e.g. 8086)
- This schema was later improved by moving the segment registers in RAM, in a segment table (array).
  - This allows a larger number of segments
  - It can organize memory closer to the logical program memory layout
  - Different segments have different permissions





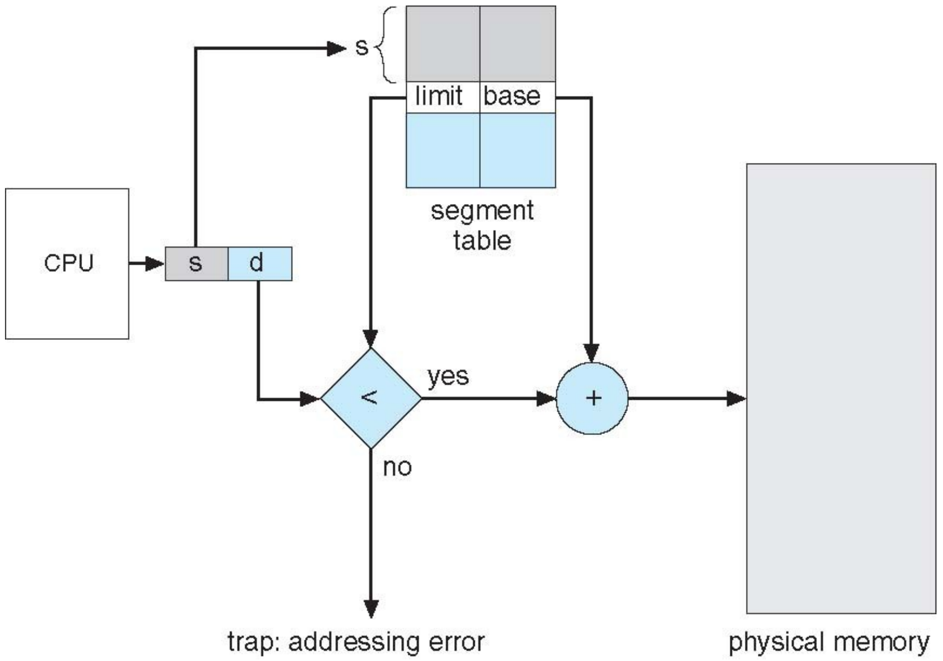
Segmentation (HW)

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**



Protection

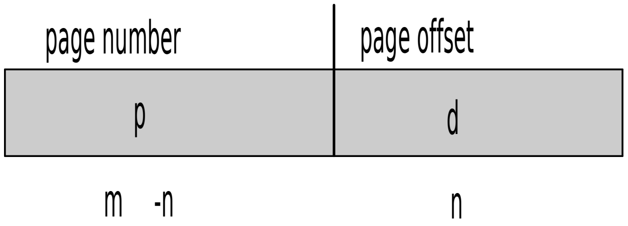
- With each entry in segment table associate:
  - validation bit = 0 ⇒ illegal segment
  - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level

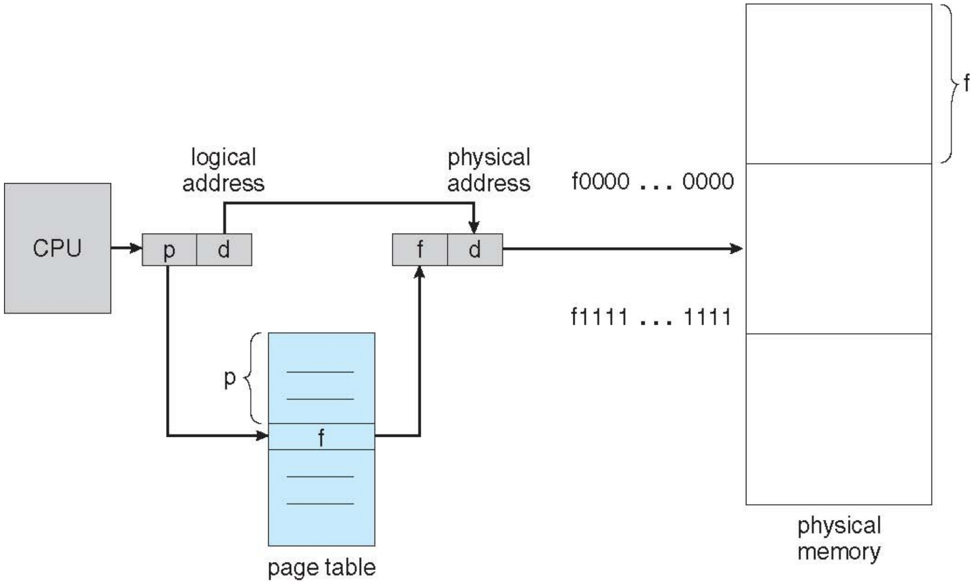
Paging (HW)

Modern way used to virtualize memory

- Divide physical memory into fixed-sized blocks called frames whose size is a power of 2
- Divide logical memory into blocks of same size called **pages**
- Set up a **page table** to translate logical to physical addresses

- Address generated by CPU is divided into:
  - **Page number** (***p***) – used as an index into a **page table** which contains base address of each page in physical memory
  - **Page offset** (***d***) – combined with base address to define the physical memory address that is sent to the memory unit





Paging (HW)

Page table is kept in main memory

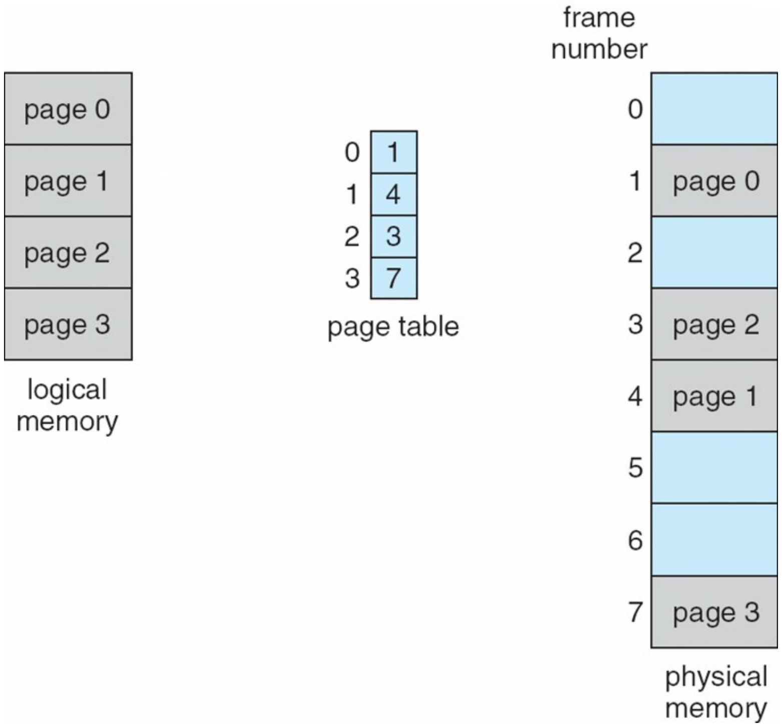
- **Page-table base register** (**PTBR**) points to the page table
- **Page-table length register** (**PTLR**) indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses
  - One for the page table and one for the data / instruction
  - The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers** (**TLBs**)

- Pros
  - no external fragmentation
- Cons
  - internal fragmentation (lowers as pages gets smaller)
  - longer memory accesses (2 RAM cycles), lessened with TLB
    - In segmentation, we stay in a set of segments, so extra-segment accesses are expensive, but controlled
  - Need space for page table (increases as page size decreases)

Paging Examples

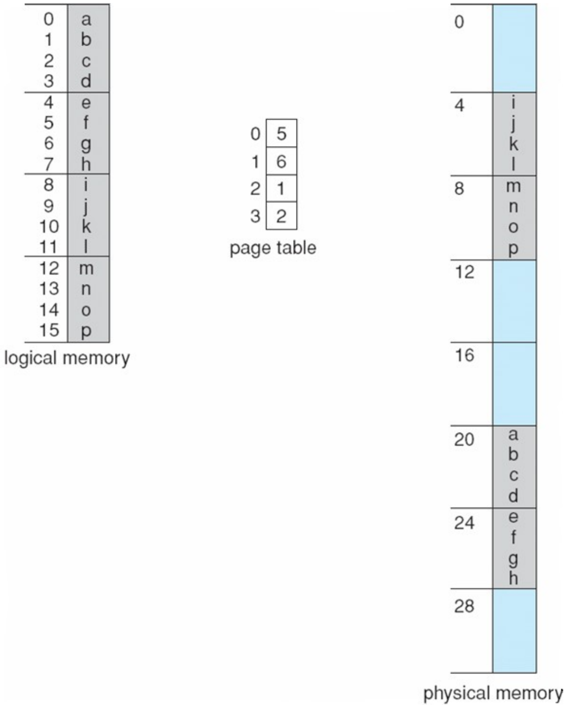
Process View

- logical addresses are contiguous
- Physical addresses are scattered



Detail

- n=2, m=4, 4 byte pages



TLB

Stands for Translation Lookaside Buffer

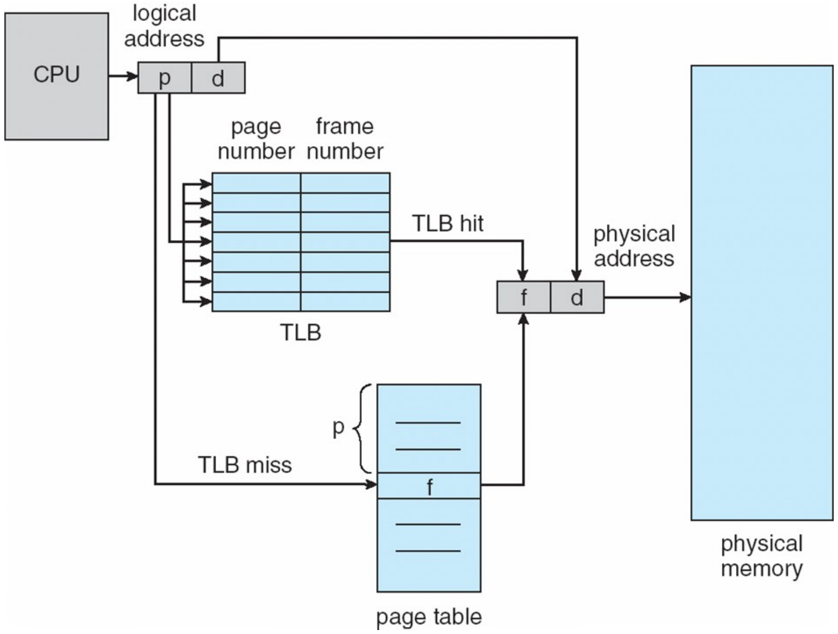
- It is an associative memory in the CPU that stores entries of the page table
- Locality principle ensures that a page accessed now is likely to be accessed very soon
- Used to cache the entries of the page table

TLB is orders of magnitude faster than RAM

Usually a rather limited number of entries

When the CPU wants to access a logical address it first looks in the TLB for the corresponding page is present

- if present, it uses the frame from the TLB
- otherwise it reads from the page table the correct entry and substitutes an old entry in the TLB
- finally it generates the right physical address



Paging Protection

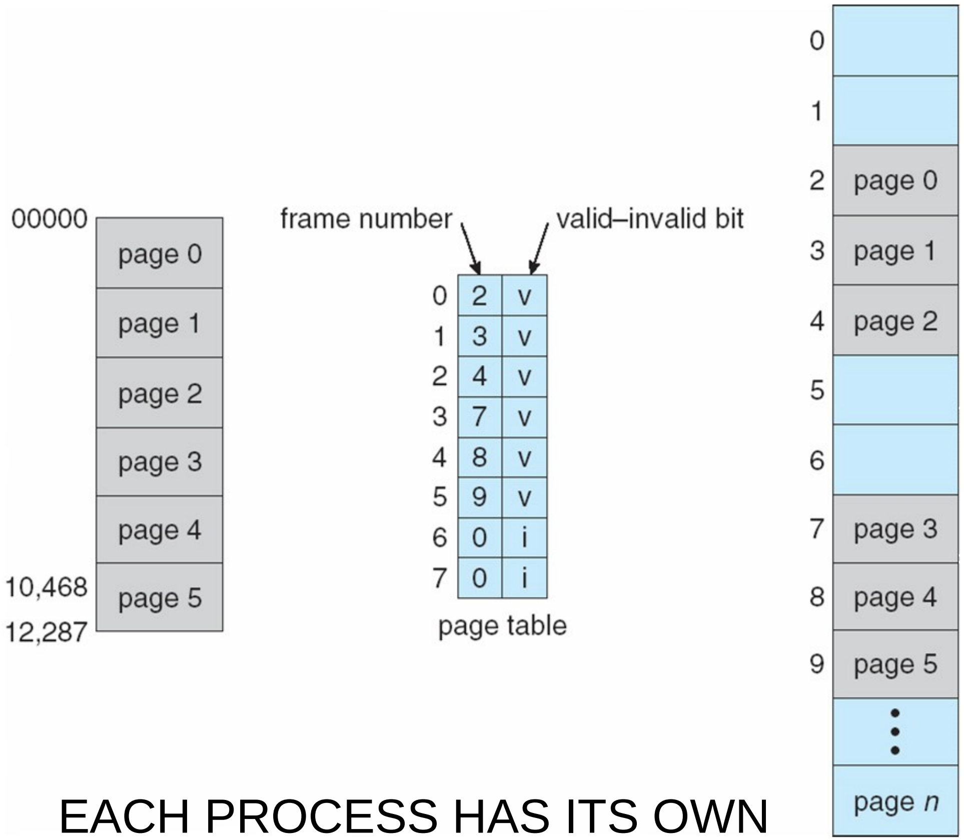
Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed

- Can also add more bits to indicate page execute-only, and so on

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
- Or use **page-table length register** (**PTLR**)

Any violations result in a trap to the kernel

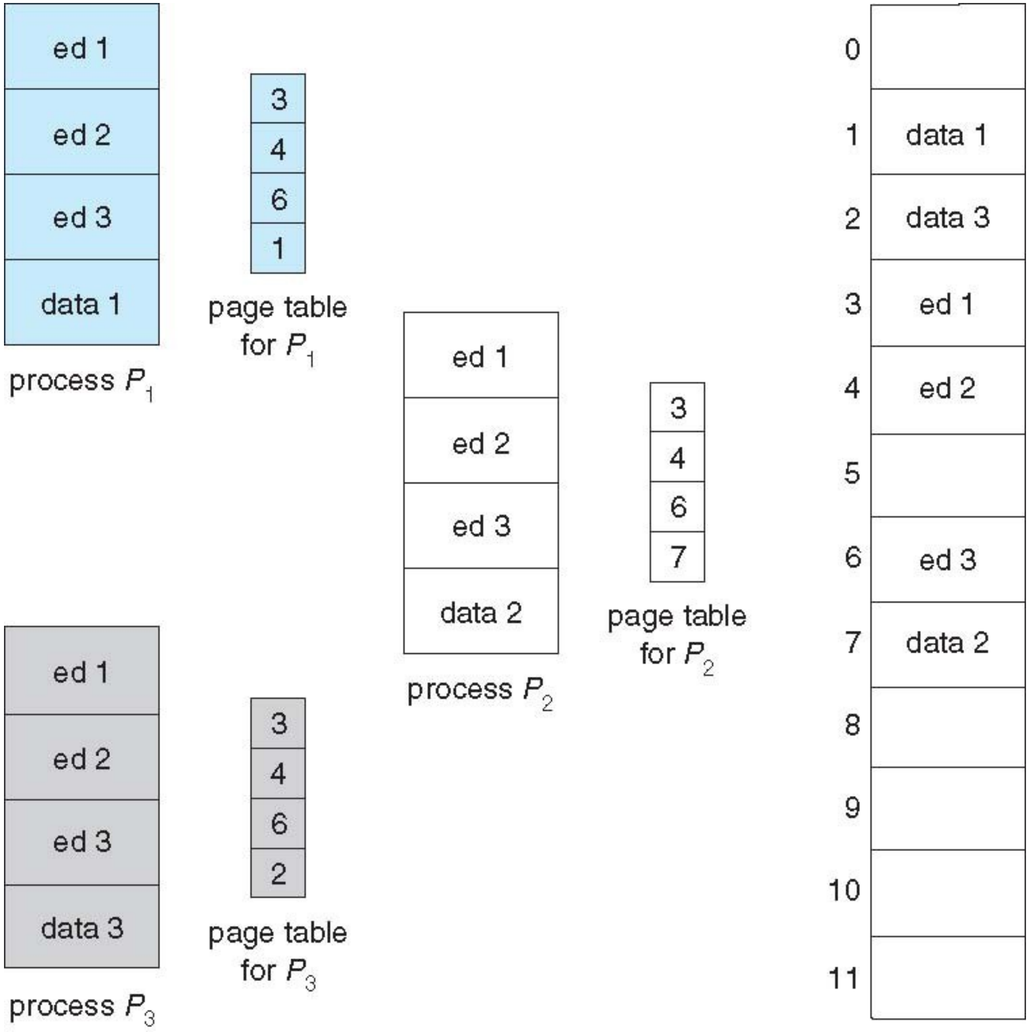


	frame number	valid–invalid bit
0	2	v
1	3	v
2	4	v
3	7	v
4	8	v
5	9	v
6	0	i
7	0	i

EACH PROCESS HAS ITS OWN PAGE TABLE!!

Shared Pages

- **Shared code**
  - One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
  - Similar to multiple threads sharing the same process space
  - Also useful for interprocess communication if sharing of read-write pages is allowed
- **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



Effective Access Time (EAT)

EAT: expected time of a memory operation

$$\mathrm{EAT} = \alpha(T_{\mathrm{ram}} + T_{\mathrm{tlb}}) + (1-\alpha)\cdot 2\cdot(T_{\mathrm{ram}} + T_{\mathrm{tlb}})$$

Computed on the statistical basis

- T_tlb: access time to TLB
- T_ram: access time to RAM
- α: hit ratio, ercentage of times that a page number is found in the associative registers; ratio related to number of associative registers

Example

- Consider α = 80% T_tlb=20ns and T_ram=100ns

  EAT = 0.80 x 120 + 0.20 x 240 = 144ns

- Consider more realistic hit ratio -> α = 99%, same timings

  EAT = 0.99 x 120 + 0.01 x 220 = 121.2ns

Hierarchical Page Tables

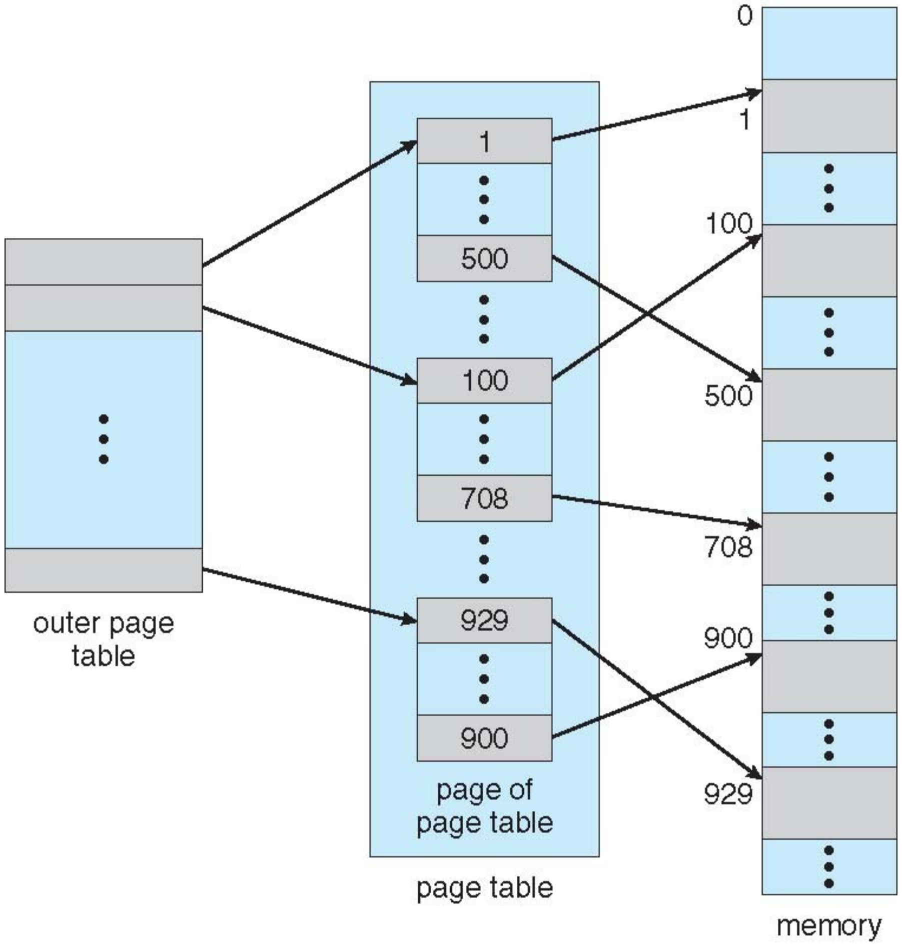
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table

Pros:

- Reduces the size of the page table
- With modern memory capacities, page table can be HUGE

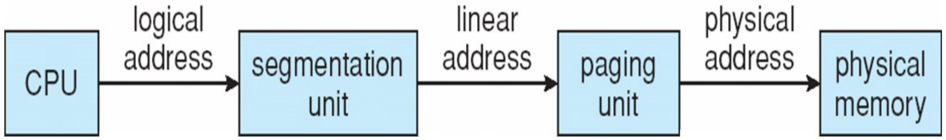
Cons:

- increases the EAT (more penalty on the misses)

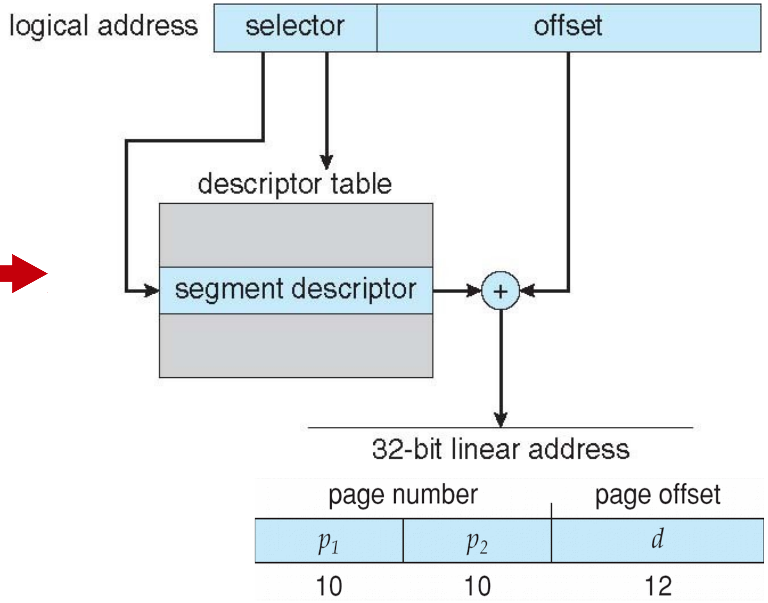


IA-32

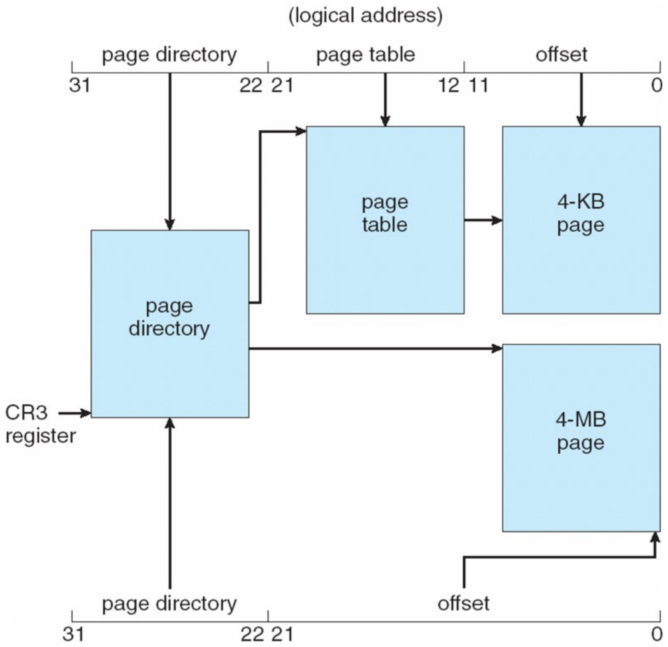
- Combines segmentation and paging



- CPU generates logical address
  - Selector given to segmentation unit, which produces linear addresses

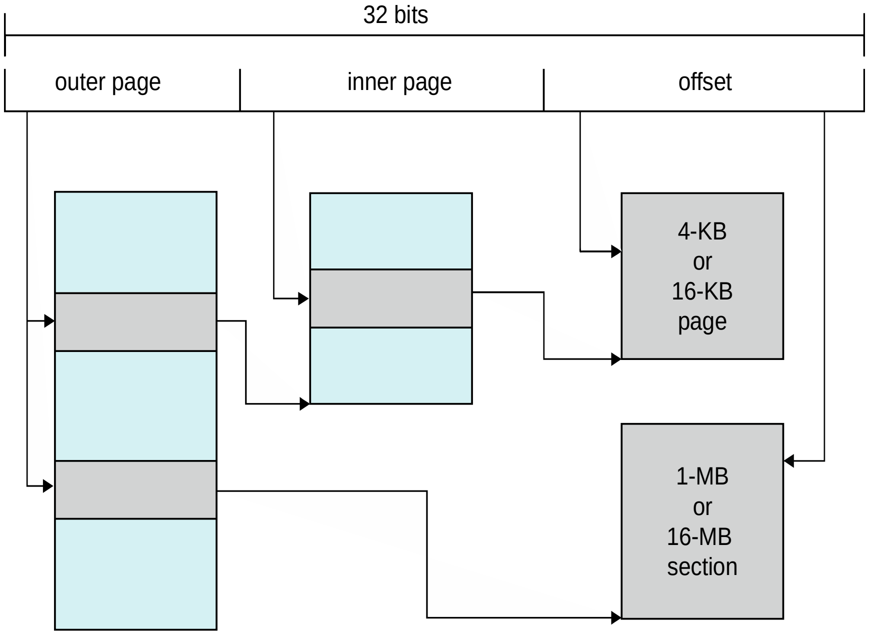


  - Linear address given to a 2level paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU
    - Pages sizes can be 4 KB or 4 MB



ARM Architecture

- Modern, energy efficient, 32-bit CPU
- 4 KB and 16 KB pages
- 1 MB and 16 MB pages (termed **sections**)
- One-level paging for sections, two-level for smaller pages
- Two levels of TLBs
  - Outer level has two micro TLBs (one data, one instruction)
  - Inner is single main TLB
  - First inner is checked, on miss outers are checked, and on miss page table walk performed by CPU



Swapping

A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution

- Total physical memory space of processes can exceed physical memory

Backing store – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images

Roll out, roll in – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed

Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped

System maintains a **ready queue** of ready-to-run processes which have memory images on disk

Swapping

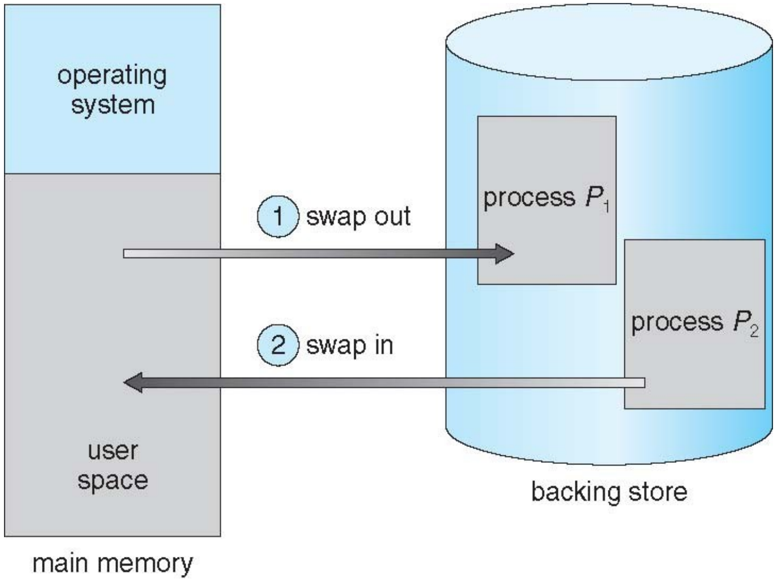
Does the swapped out process need to swap back in to same physical addresses?

Depends on address binding method

- Plus consider pending I/O to / from process memory space

On modern OSes

- Swapping normally disabled
- Started if more than threshold amount of memory allocated
- Disabled again once memory demand reduced below threshold



Swapping and Context Switch

If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process

Therefore, context switch time can be very high

100MB process swapping to hard disk with transfer rate of 50MB/sec

- Swap out time: 2000 ms
- Plus swap in of same sized process
- Total context switch swapping time: 4000ms (4 seconds)

Can be reduced by reducing the size of the memory swapped – by knowing how much memory really being used

- System calls to inform OS of memory use via `request_memory()` and `release_memory()`

Swapping and Context Switch

Other constraints as well on swapping

  - Pending I/O – can’t swap out as I/O would occur to wrong process
  - Or always transfer I/O to kernel space, then to I/O device
    - Known as **double buffering**, adds overhead

- Standard swapping not used in modern operating systems
  - But modified version common
  - Swap only when free memory extremely low