- Addresses program generates = *virtual addresses*
- Addresses in memory = *real* or *physical* addresses
- Number of virtual addresses far exceed the number of real addresses
- Must mean that the real address space must be shared (reused) during the execution of a program
- *Dynamic Address Translation* : when virtual addresses translated to real addresses as process executes
- The concept of a *virtual machine* : machine larger than the real machine, sustained by the use of *auxiliary* storage (disks, tapes etc)
- At any given time, possible to have a diverse mix of many users programs in memory at the same time

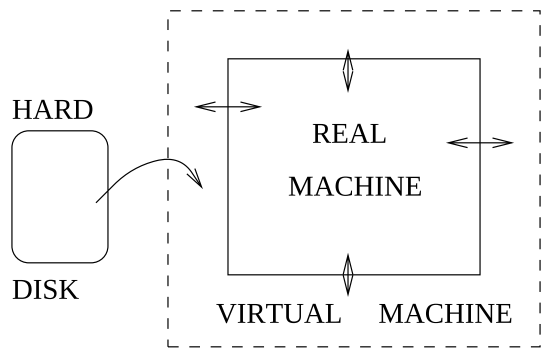

Figure 1: Virtual to physical address mapping

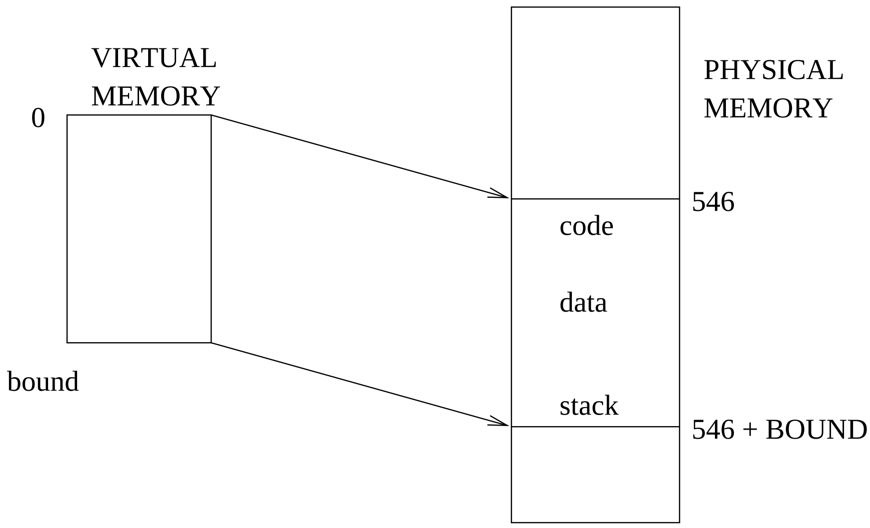

Figure 2: Virtual to physical address mapping

- In *base and bound* systems, OS can relocate program *in the program's back* - transparency - a mechanism of indirection
- Only the Os gets to change the *base* and *bounds*
- Implementation cost : 2 registers, adder and comparator
- + : simple and fast
- - : hard to share between programs ; hard to grow address space, complex memory allocation (may need large amount of shuffling)

Solutions ?

1 Segmentation

- A **segment** is a region of logically contiguous memory
- Have a **table of** *branch and bounds*

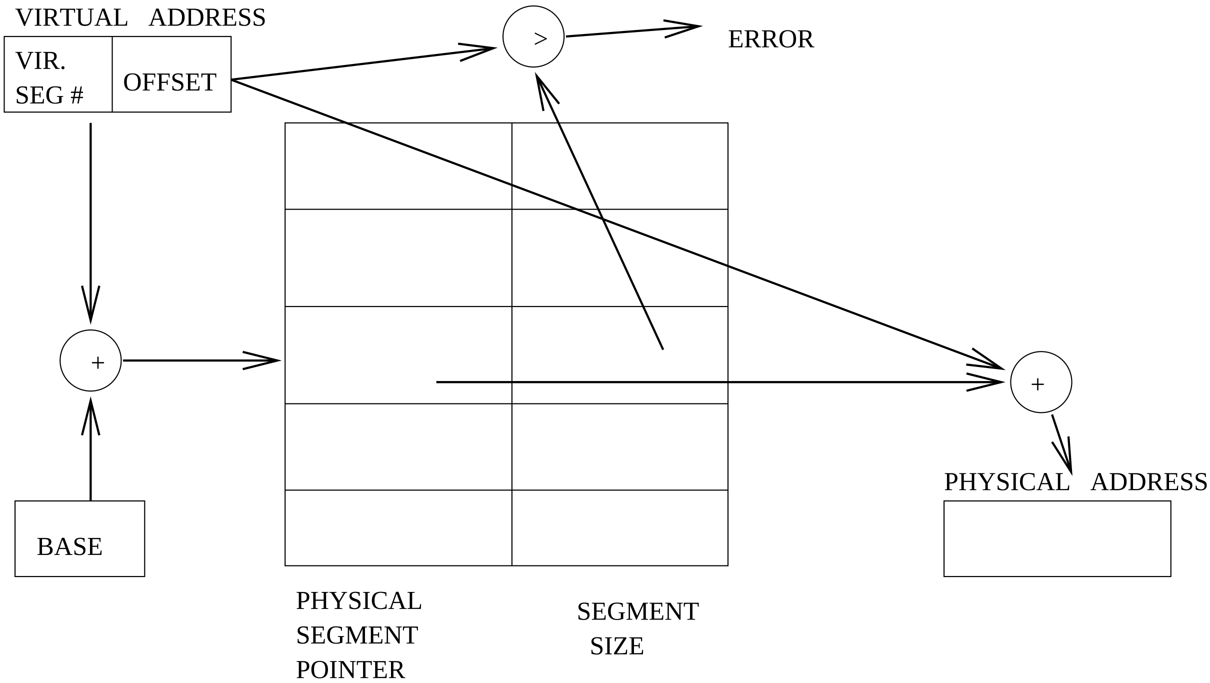

Figure 3: Segment based mapping

- Program issues a (binary) *virtual address 10011* which translates to location 19. However, the virtual address is broken into 100 (which is the segment number) and an offset 11. The segment number is added to a base register (which contains the base address of the segment table), and is used to reference a location within the *segment table*.
- The *real/physical address* is obtained by adding the *segment table* entry with the offset. The *segment table* entry is the primary storage address.

Consider the following with 2 bit segment ID, and 12 bit segment offset.

virtual segment no.	physical segment start	segment size
code	0x4000	0x700
data	0	0x500
stack	0x2000	0x1000

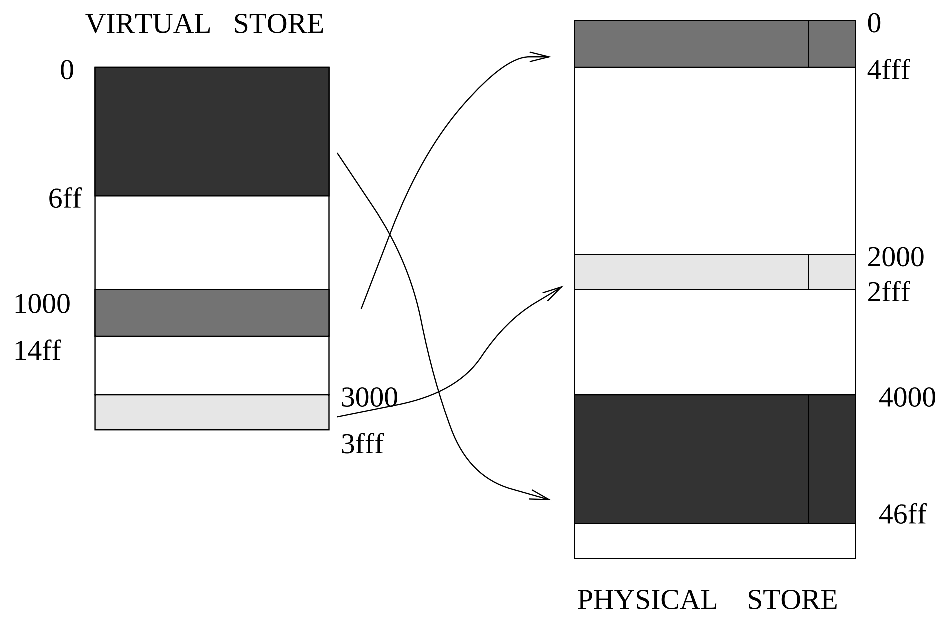

Figure 4: Segment based mapping

- Possible to have a protection mechanism for segments, to control process access
- *Four* main categories of *access* : *Read, Write, Execute and Append* - can combine modes to obtain more complex access strategies.
- A more detailed *segment table entry* contains : *segment present*, *secondary memory address*, *segment length*, *access control bits*, *address of segment if in primary memory*
- *missing segment fault* : segment NOT in main (primary) memory (load segment from 2ndary memory)
- *segment overflow exception*: displacement is greater than segment length
- *segment protection exception*: required operation not permitted on segment

Segment Present bit	2ndary mem add.	length	R	W	E	A	primary address

Segment Present bit	2ndary mem add.	length	A 1	A 2

Figure 5: Segment based mapping

- Segments can be of *arbitrary size* - hence a segment used to represent a *dynamic structure* can *grow* as the structure grows. Also, a segment may *shrink* in size over its life time (same reasoning as growing)

- Possible to *share* segments between processes. This is achieved by simply pointing the segment table addresses of the processes to the same position in main (primary) memory.

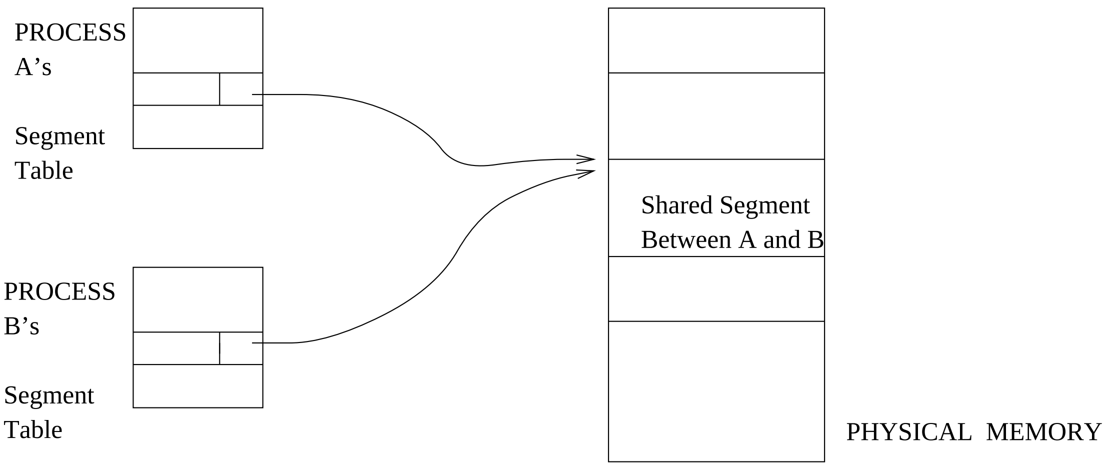

Figure 6: Segment based mapping

- Segmentation + : *Efficient for sparse address spaces*
  Easy to share segments
  Support for dynamic data structures which can grow or shrink over time

- Segmentation - : *Complex memory allocation*
  Still need shuffling to coalesce free space - if no free space for a segment

- Need to simplify - so how do we make memory allocation simple and easy ?

2 Paging

- Segments can be of *variable* lengths, whereas pages are *fixed* length - memory allocated in *chunks* of storage
- Virtual address = *(p,d)*, where *p*: page number in virtual storage and *d*: displacement within the page
- Pages are placed in primary memory in blocks - *page frames* of equal size as incoming pages
- page frames are integral multiples of page size (in primary memory - referenced by a page table pointer
- Each address space has its own page table

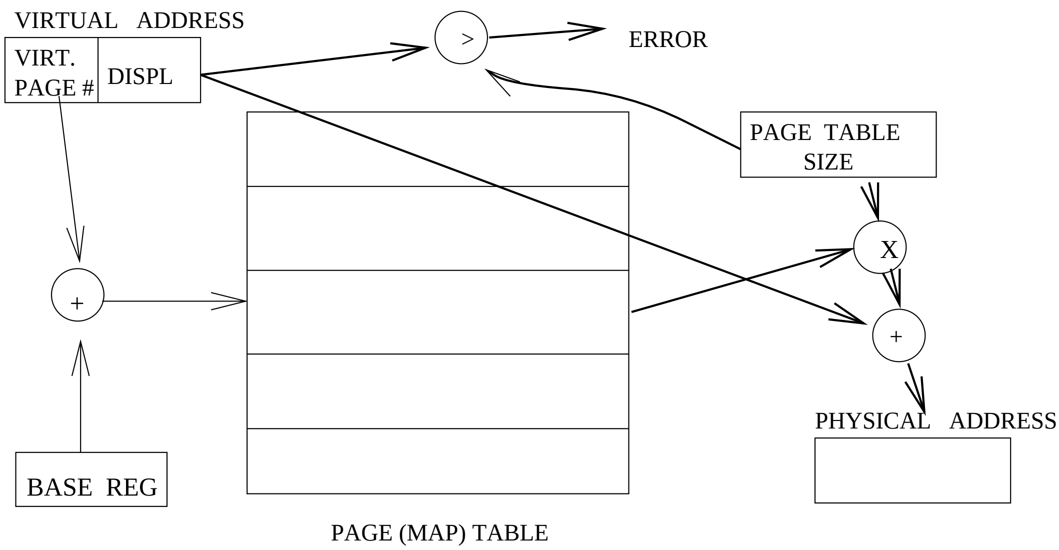

Figure 7: Paging - direct translation

- What happens when required (referenced) page is not in memory ?
- Need to get it from *secondary storage* - the page table entry must provide an indication of where the page is and how to obtain it from secondary storage (similar to a segment table entry)

Page Residence bit	2ndary storage add:	Page Frame Number

Figure 8: Page table entry

- Many ways of translating *virtual* page address to *real* page address : *Direct* (mentioned already), *Associative* mapping and variations
- Direct method is too slow, because *page table* can be quite large and must be maintained in primary memory, and comparison must be performed between ALL entries one at a time
- Alternative is to keep the page table in high speed memory called *cache* (will come to this later)
- Another alternative is to find a different way to perform the comparison : *Associative Mapping* - page table now in a content addressed associative storage

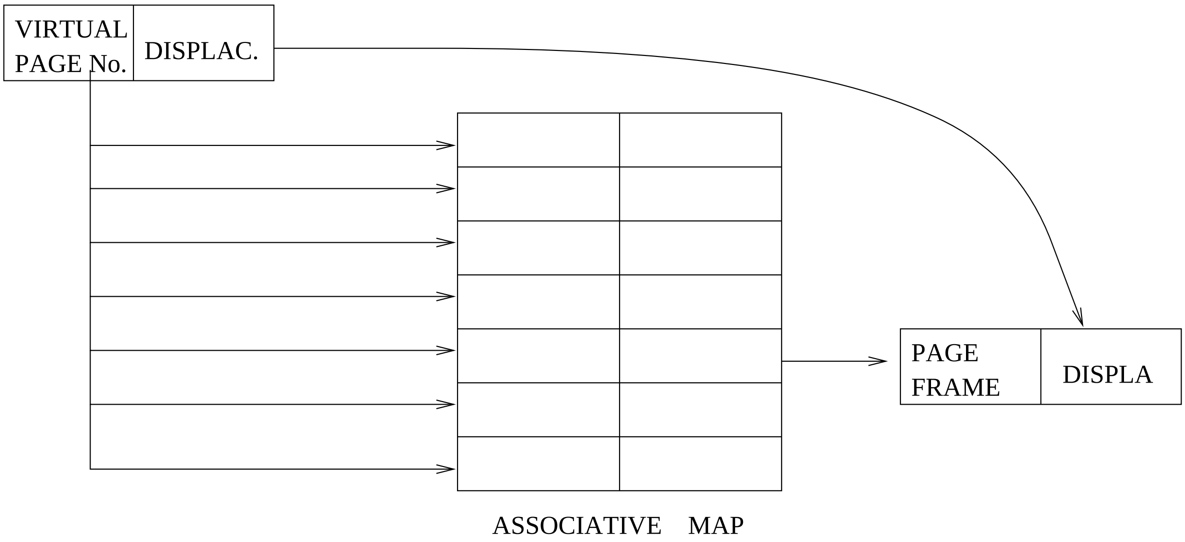

Figure 9: Associative Mapping

- When a virtual address v=(p,d) is issued, every entry in the associative storage is searched simultaneously for page p
- Much quicker, as do not need to perform a search sequentially - if a match is found, the real address can be calculated
- However, if still slow, possible to combine direct mapping with associative mapping
- Also possible to have sharing in a paging system, where multiple page frame references in page tables point to the same area of real (physical) storage

- Paging : + : *simple memory allocation, easy to share*
  - : *big page table if sparse address space* $\equiv$ space wastage
- What's the solution ? : *allows simple memory allocation, easy sharing* **and** *efficient for sparse address spaces*
- Combine *paging and segmentation* = multilevel translation
- Use tree of tables, *Lowest level is page table - Higher levels = segmented*
- Segments are restricted to be a multiple of page sizes (though this multiple is not fixed)
- Most architectures today implement some flavour of this

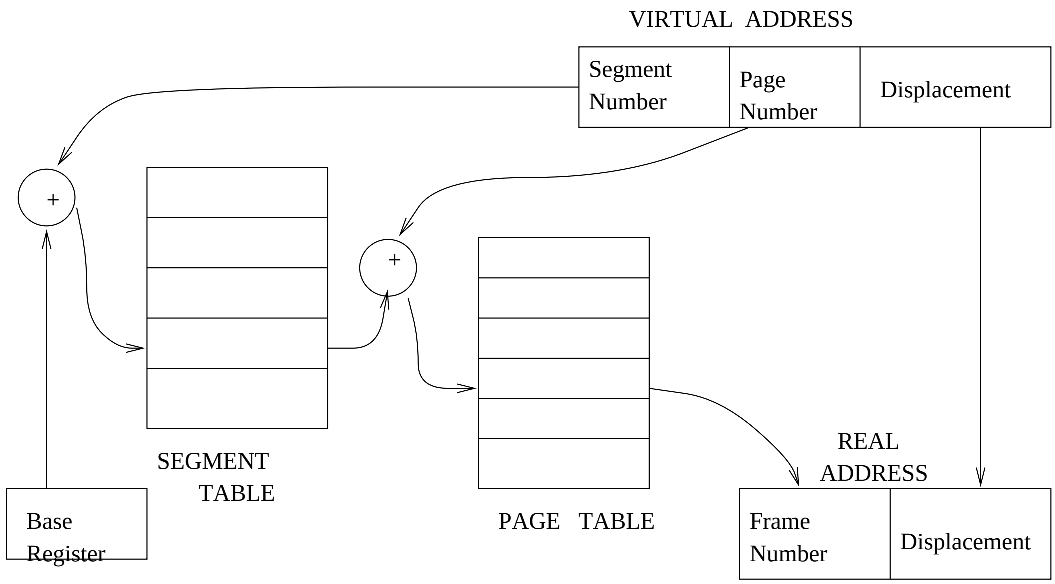

Figure 10: Paging + Segmentation

- Where is page table kept ? Where is segment table kept ?
- In primary store - if too large - too much wasted space in primary memory - overhead (as storing tables NOT pages or segments themselves) - so what do we do ?
- Also, possible to put page tables in a special segment that is only accessible to OS
- Use a *Translation Lookaside Buffer* (TLB) (via cache) : **Idea** : Make frequent case efficient - infrequent path doesn't matter all that much
- Cache is a fast memory (closer to the CPU than primary memory - and implemented using high speed registers)
- What is this concept ? and more importantly, why does it work better ?
- One of the fundamental concepts in computing : **locality of reference** : pages or segments accessed last are **MOST likely** to be accessed again - so place them in the cache

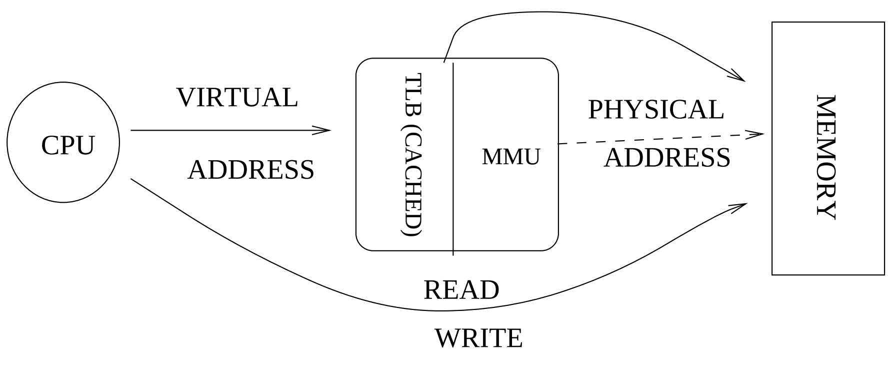

Figure 11: Paging + Segmentation

3 How do we replace pages or segments in cache ?

- **Fetch** and **Replacement** policies
- Consider this scenario : Primary (main) memory is full. No more vacancies. BUT need to store a page in it - need to get it from secondary store ... *where do we put it ?*
- Answer : *We don't*, we've got to take another page out to make room (c.f. CD or tape stacking problem)
- *Segment placement* more difficult - as *segments* can be of variable size
- Most obvious : **Random** : Take some page out at random and replace with incoming page *May replace the newest page !*
- More obvious : **First In First Out** : Take out page that has been in primary store for the longest period *Most referenced page is oldest!*

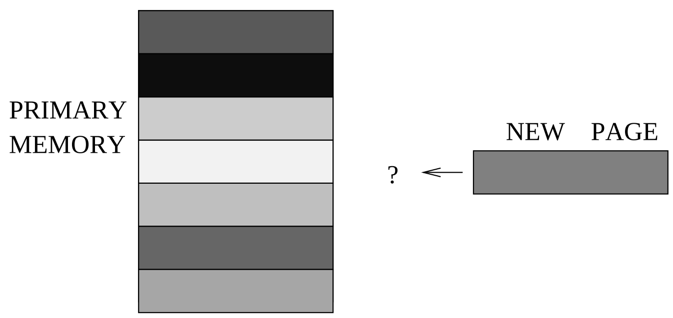

Figure 12: Page Replacement - where to place ?

- **Least Recently Used** : Get rid of page not used over the recent past (locality of reference) - Pages need to be timestamped
- **Least Frequently Used** : Replace page according to frequency of usage
- **Not-Used-Recently** : When page was last modified - has a *dirty bit* - approximates LRU
- **Second Chance replacement FIFO** : Have a queue of processes - and load in a particular order when space become available - based on the presence of a *referenced bit* being set. Idea = active pages will have referenced bit set, and move to the head of the queue
- **Locality of reference** : *temporal or spatial* : temporal (locality in time e.g. loops subroutines, stacks), spatial (locality in terms of neighbours - in space, e.g. arrays, sequential code generation).
- *Locality implies* : Processes tend to access some pages more than others (these are usually adjacent to each other (in virtual address space), or are related temporally), *subset of most referenced pages* for a process - hence the **working set model**

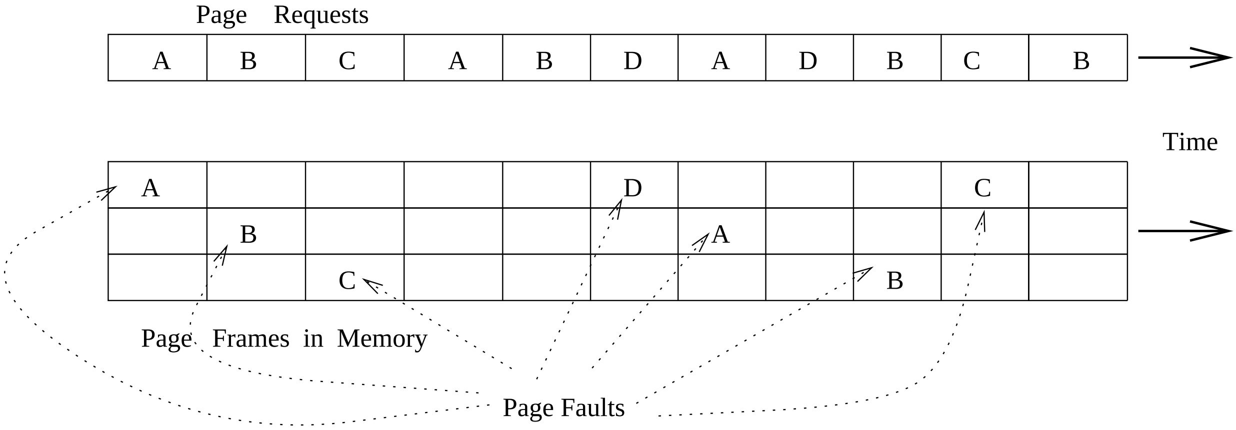

Figure 13: Page Replacement - FIFO ?

4 The Working Set

- A set of pages a program *frequently* references
- To run efficiently, most of the program's working set must be in primary memory - else we get **thrashing** (program frequently requests pages from primary storage)
- Whether to add a new process would imply checking whether most of its working set can be maintained in primary memory
- Working set could change over the process lifetime, as it changes phases
- The **working set size** if a *major* design criteria for the OS designer

5 Page Fault Frequency

- Page fault : when request page not in primary memory
- Wait till something fails : then adjust !

- *Use page fault frequency* to adjust working set size (trying to satisfy many constraints in system) - i.e. multiple processes and page faults
- *Demand Paging* : Load pages only when requested - *slow* - but *good if program changes behaviour rapidly*
- *Anticipatory or Pre- Paging* : predict behaviour of program (based on locality of reference perhaps)

6 Some Auxiliary Issues

- *Page release* : user process automatically says - this page not needed (from its working set)
- Size Matters !! : *Size of page frames and pages* : *Small size* $\equiv$ *lots of diversity within store, as can store many different pages simultaneously*, however, *too many requests to disk if need to load small units*
  Large size $\equiv$ *not all of the contents of a page may be used*
- Should pages be the same as *disk blocks* ? 512 K to 1024 K on Unix (or maybe multiples of them) ?
- Possible that not all of a page is used : leads to fragmentation - so small the page size lesser the fragmentation

Remember : Programs spend 90 a heuristic to design your OS).