University College Dublin
An Coláiste Ollscoile, Baile Átha Cliath

# Operating Systems
# Virtual Memory

Dr. Vivek Nallur (vivek.nallur@ucd.ie)

## Virtual Memory

## Virtual Memory:Characteristics of paging and segmentation:

- the address space of a process may be broken up into pieces (pages or segments)
- memory references within a process are logical addresses, dynamically translated into physical addresses

## Virtual Memory:Characteristics of paging and segmentation:

- by virtue of the page/segment table, pages/segments of a process need not be contiguously located in main memory
- <u>last but not least:</u> it is not necessary that all pages/segments of a process be simultaneously in main memory during execution: part of the process may be disk

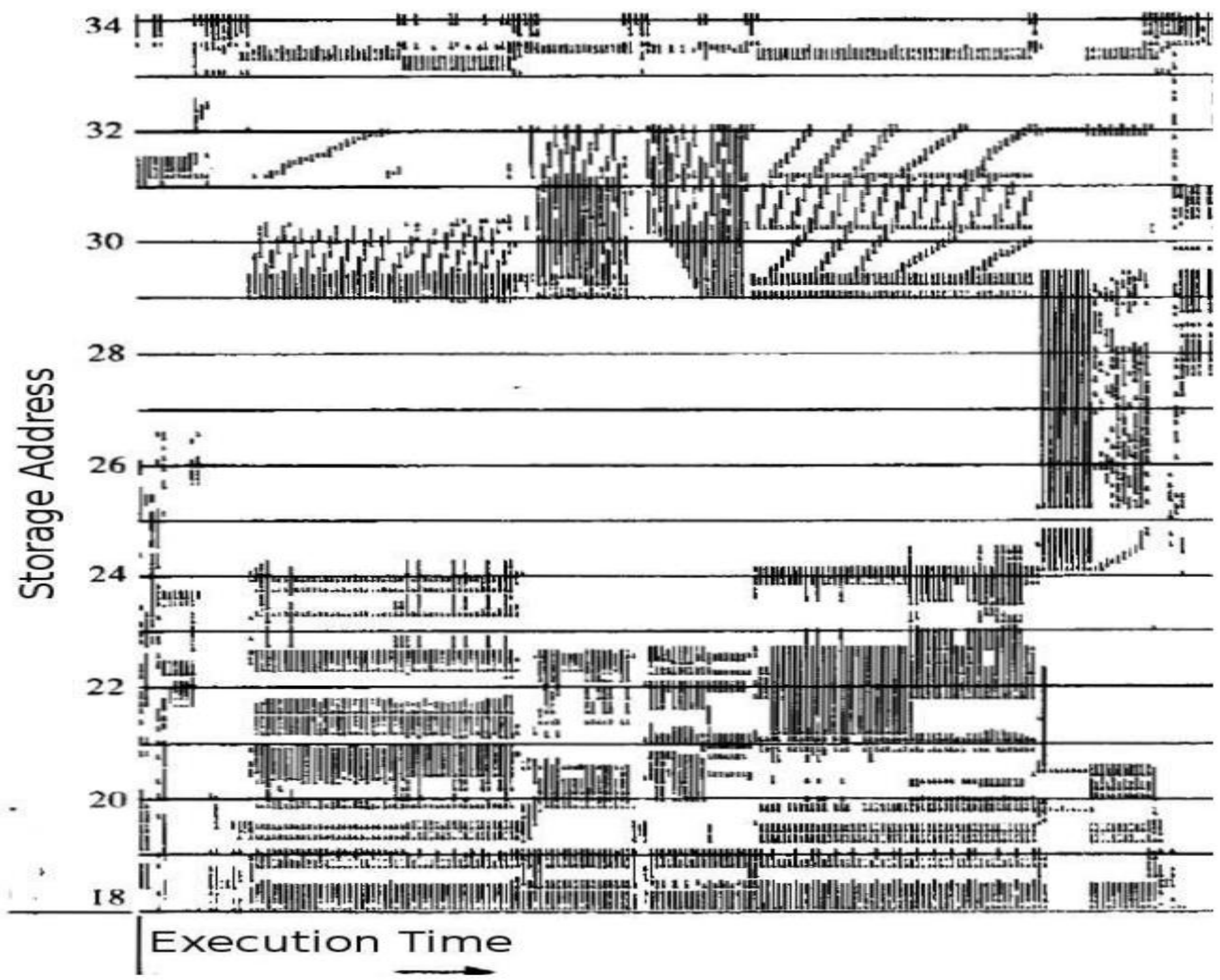
## Virtual Memory

- Virtual memory (VM): is an illusion supported by system hardware and software that a process has a vast and linear expanse of available memory (much bigger than the main memory unit)
  - with VM, the main memory can be seen as a cache for the disk
  - logical addresses are also called virtual addresses in the context of VM

## Real Programs and VM

- The VM scheme is appealing: is it practical as well?
- An examination of a real program shows that it can usually be divided into two parts:
  1. parts frequently needed
  2. parts rarely or never needed:
  - code handling unusual error conditions
  - code handling certain options and features rarely used
  - allocation of more memory than strictly needed (arrays,tables. . . )
- Also, memory references within a program tend to be clustered (principle of locality of references)

# Example: Locality of Memory References



## Real Programs and VM

- Therefore, it is possible to make intelligent guesses about which pieces of the address space of the process will most likely be needed in the near future
  - if guesses are good, then a system using VM will perform efficiently
- If guesses are poor, the system will suffer from thrashing
  - it will spend too much time swapping processes’ pieces between memory and disk, rather than executing instructions
  - disk accesses are much slower than memory accesses

## Virtual Memory Features

- VM is commonly implemented by demand paging
  - when a program is loaded, the OS brings into main memory only a few pages of it (including its starting point)
  - further pages are then brought to memory or swapped to disk as needed
  - the resident set is the portion of the process that is in main memory at a given time

## Virtual Memory Features

- VM system supported by hardware: paging mechanism, which generates page faults
    - when pages in disk are referenced
    - software: page swapping management (OS algorithm)

# Page Table with Demand Paging

- Additions to page table entries:
- present bit, set to one if the piece is in main memory
  - an interrupt (page fault) is generated when a reference is made to a piece not present in main memory
- modified bit, set to one if the piece has been altered since it was last loaded into main memory
  - if not modified: it does not need to be written to disk when it has to be swapped out
- control bits:
  - r/w page: a read-only page can be shared by two or more processes
  - kernel/user page: kernel pages can be forced to be in memory at all times

## Page Table with Demand Paging

- main entry:
- if page in memory: frame number
- if page in disk: address in disk, or index to a table
    - referenced in the PCB— used to obtain its address

## VM Advantages

- Programs not constrained by the physical memory space
- they can be as large as the virtual address space will allow :
  - 1970s: 32-bit virtual addresses → 4 GB of memory
  - 2000s: 64-bit virtual addresses → 16 EB of memory
- in both periods, that amount of real memory would cost millions of euro.
- Also in size, 16 EB in modern 4 TB hard drives could be placed end to end 467.2 km) and cover the whole of Beijing Subway system with 2.2 km to spare.

# VM Advantages

- Better multiprogramming: more processes can be maintained in memory at any given time
  - more likely that one of these processes will be in ready state (remember $1 - p^n$ CPU utilisation formula)
- Less I/O is needed to load a program or to swap it
  - increase in CPU utilisation and throughput

## Page Fault Procedure

- The page-fault procedure requires three major steps:
  1. service the page-fault interrupt
     - trap to the OS
     - save registers and process state
     - check if the page reference is legal and determine its location in the disk
  2. read in the missing page
  3. restart the process

## Page Fault Procedure

- The page-fault procedure requires three major steps:
  1. service the page-fault interrupt
  2. read in the missing page
     - issue a read of the missing page from the disk to a free frame
     - since this is an I/O operation (slow), another ready process is dispatched to the CPU
     - eventually, an interrupt from the disk is issued (I/O completed)
  3. restart the process

## Page Fault Procedure

- The page-fault procedure requires three major steps:
  1. service the page-fault interrupt
  2. read in the missing page
  3. restart the process
  - an interrupt from the disk indicates that the page is in memory
  - the process’s page table is updated to show that the desired page is now in memory; the process can now be switched to the ready state
  - eventually the process is scheduled again, and resumes execution at the instruction interrupted through the page fault

## Page Faults and Performance

- It is important to keep page faults to a minimum: why?
- If the probability of a page fault (or page fault rate) is p ($0 \le p \le 1$), then the effective memory access time ($T_e$) is

$$T_e = p \times T_f + (1 - p) \times T_a$$

$T_f$: page fault time ( time to process a page fault)

$T_a$: memory access time

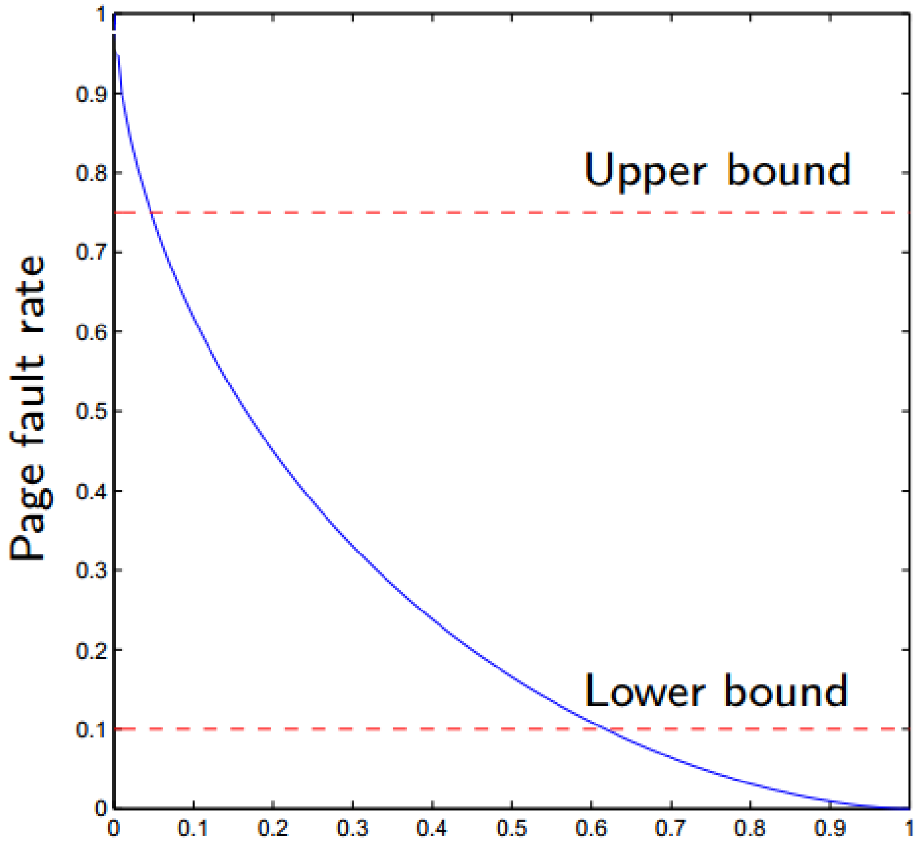
100 x p: % of memory access that raise a page fault.

## Page Faults and Performance

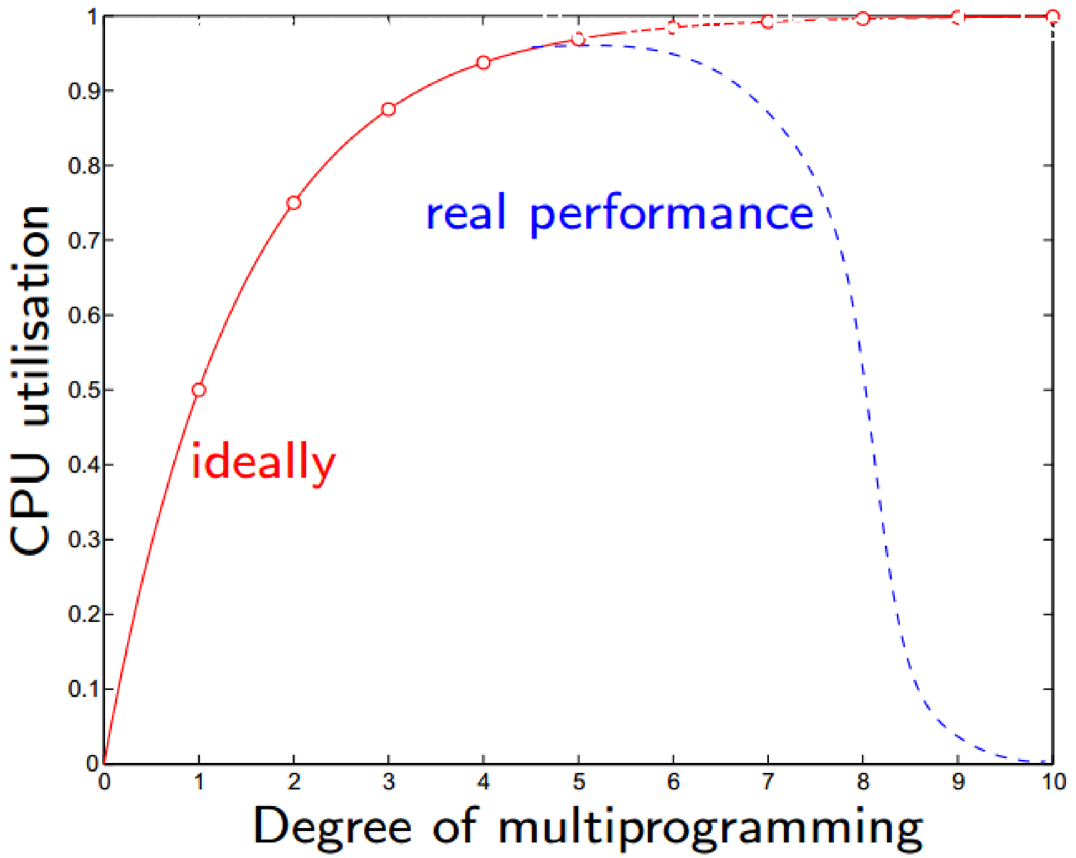
$$T_e = p \times T_f + (1 - p) \times T_a$$

- Typically, $T_f$ can be 5 orders of magnitude higher than $T_a$ (e.g. 10 ms vs 100 ns)
  - therefore, even low probabilities of page fault may raise $T_{ae}$ noticeably

# Page Faults and Thrashing



## Thrashing and Multiprogramming



## Thrashing and Multiprogramming

- To avoid this, load control measures are needed. Examples:
  - only execute processes whose resident set is large enough
  - adjust effective degree of multiprogramming so that mean time between faults equals target (suspend processes as needed)

## Memory Management in VM

- In a system with VM, the OS must mainly deal with two memory management issues:
  1. Replacement policy
  2. Resident set management

## Memory Management in VM

1. Replacement policy

- what happens when a page fault occurs and there is no free frame to swap a page in?
- the OS must select a frame for replacement (to be swapped out to disk) when a new page must be brought in
- restrictions are usually placed on the page replacement policy:
  - much of the kernel and control structures of the OS are held on locked frames (cannot be replaced)

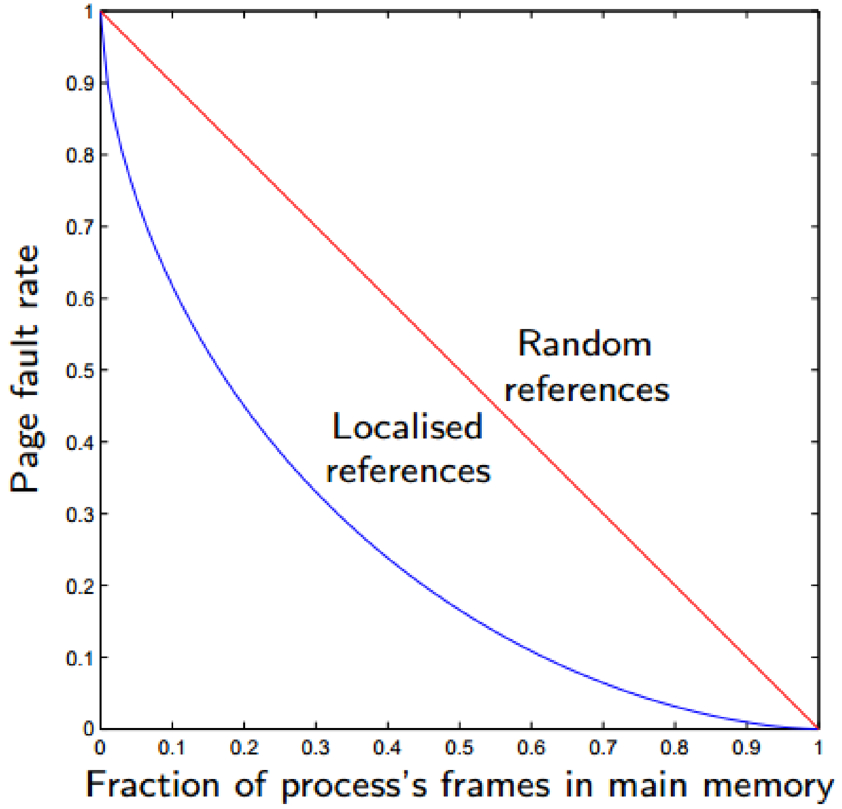
## Memory Management in VM

### 2. Resident set management

- choice of dynamic or static number of frames for each active process
- choice of replacement type allowed; examples:
  - limited to pages of the process that caused the page fault
  - encompassing any frame in main memory

## Replacement Policy and Locality

- **A good replacement policy should exploit the principle of locality of references**
- **If memory references were random rather than localised, we would not be able to pin down the working set efficiently:**



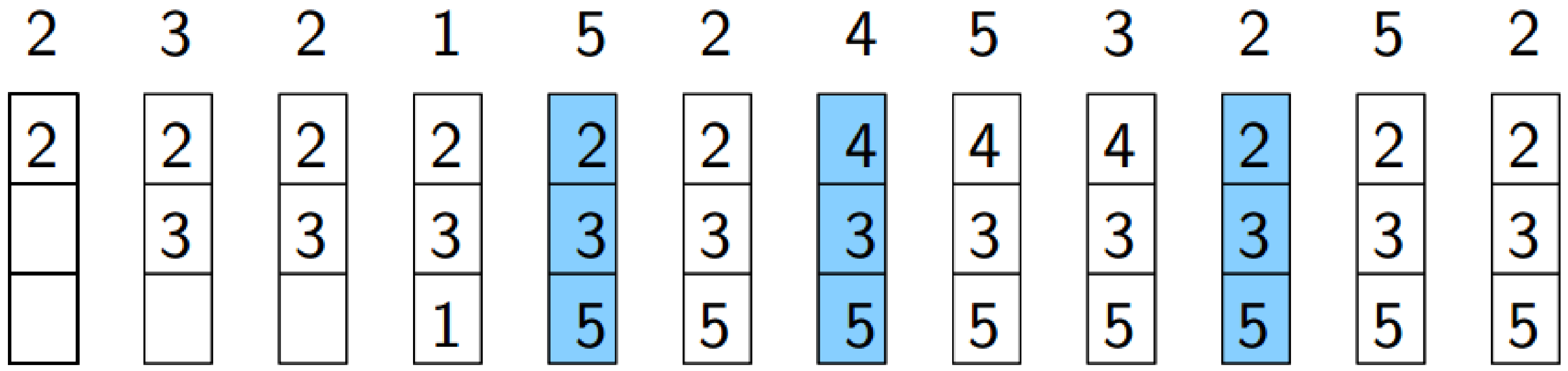
## Replacement Algorithms

1. Optimal
2. LRU (least recently used)
3. FIFO
4. Clock policy

## Replacement Algorithms
Optimal

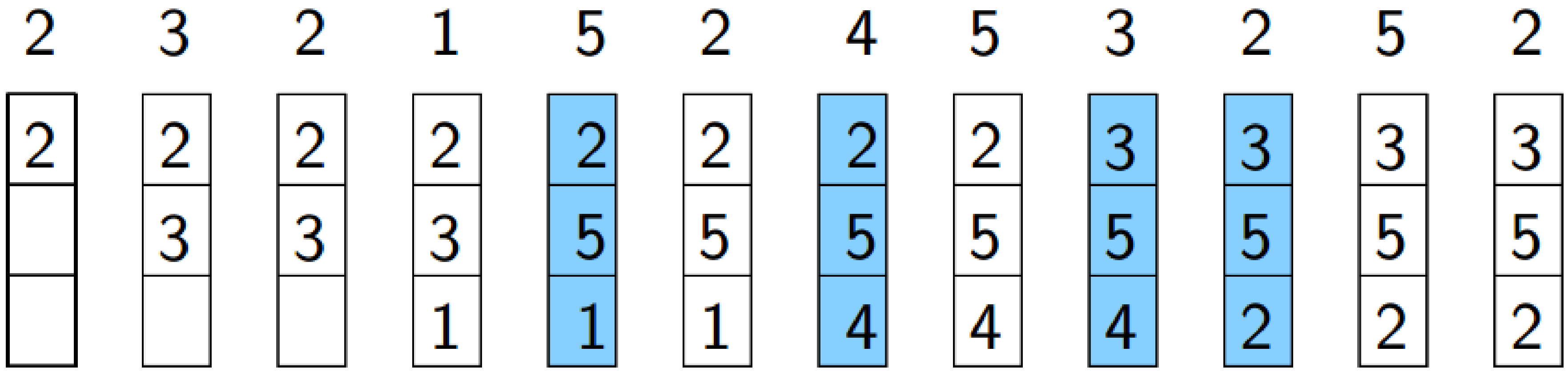
- Optimal
  - replaces the page that will not be referenced for the longest period of time
  - minimum number of page faults, but impossible to implement (knowledge of future events required)
  - standard yardstick used to gauge other algorithms
  - toy example: consider the page address stream 2 3 2 1 5 2 4 5 3 2 5 2 by a process with static resident set of just 3 frames

## Replacement Algorithms
## Optimal



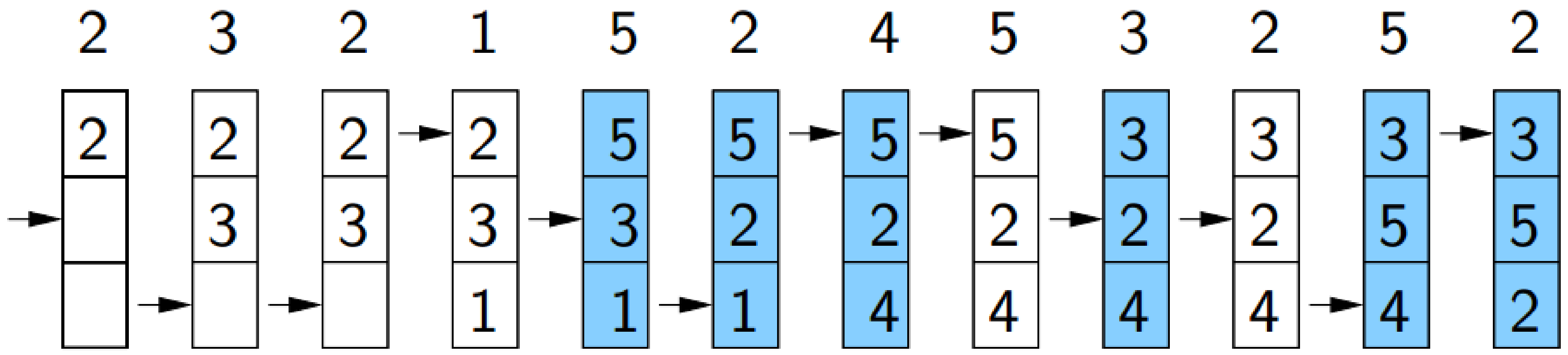
## Replacement Algorithms
## LRU (least recently used)

- replaces the page that has not been referenced for the longest period of time
- by the principle of locality: it is likely that this page will not be referenced in the near future either
- almost as good as the optimal policy, but difficult to implement (overheads associated to time keeping)


2 3 2 1 5 2 4 5 3 2 5 2
2 2 2 2 2 2 2 2 3 3 3 3
3 3 3 5 5 5 5 5 5 5 5
1 1 1 4 4 4 2 2 2

## Replacement Algorithms FIFO

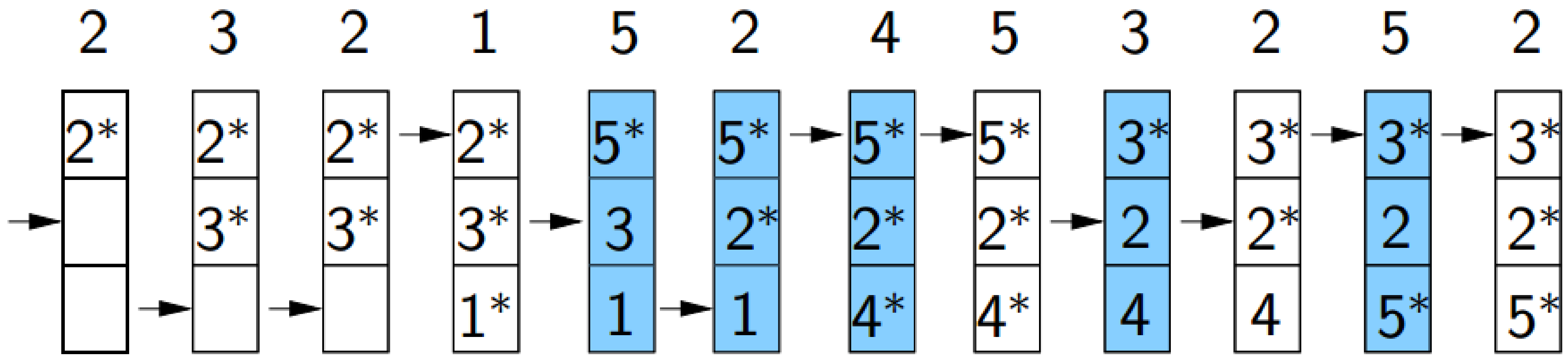
- frames traversed as a circular buffer, triggered by replacements
- pages are removed in round-robin style using a pointer (→)
- rationale: a page fetched long ago may be now out of use (when main memory is composed by many frames)
- simple to implement, but some replacements will not be good


2 3 2 1 5 2 4 5 3 2 5 2
2
2 3
2 3
2 3 1
5 3 1
5 2 1
5 2 4
5 2 4
3 2 4
3 2 4
3 5 4
3 5 2

# Replacement Algorithms Clock policy

- variant of FIFO that accounts for page use
- hardware keeps a use bit (reference bit) per page
  - when page referenced or first loaded, use bit is set to 1
- replacement is made by OS:
  - frames traversed as a circular buffer as in FIFO policy
  - first frame encountered with use bit set to 0 is replaced (logic: it has not been used for while)
  - while traversing, frames with use bit 1 get it set to 0

# Replacement Algorithms
## Clock policy



## Page Size Considerations

- Page size is invariably a power of 2, but how do we select it?
- No single best answer:
    - small page size leads to large page table
        - example: for a VM of 4 MB (222) we would have 4096 pages of 1024 bytes, but only 512 pages of 8192 bytes
    - memory is better used with smaller page sizes
        - on average, half of the last page is wasted
    - I/O transfer time is small compared to seek and latency, which favours larger pages
        - however locality improves with smaller page size (higher resolution)
- The historical trend is towards larger pages
    - 1990: typically around 4096 bytes
    - 2000's: OS's developed concept of Huge Page size
        - Typically in consumer level 2MB or 4MB
    - 2010's : This concept in enterprise systems can go up to 256 MB

## Page Table Structure

- Issue: page tables can take a big portion of memory
  - $2^{m-n}$ entries required for an m-bit long virtual address with n page offset bits
  - example: with $m = 32$ and 4 KB page size → $2^{20} \approx 10^6$ entries
  - this problem is even more severe with modern 64-bit addresses
- Solutions:
  - hierarchical paging
  - inverted page table

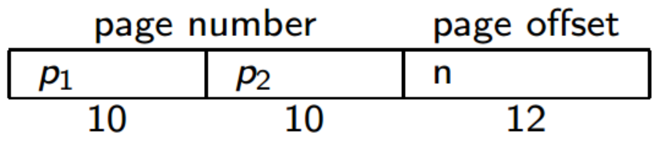
## Hierarchical Paging

- Two-level page table: the m − n page number bits are divided into two sections

1. the first points at an entry in the outer page table, which gives a frame corresponding to a page of the page table proper
2. the second points at an entry within that page of the page table

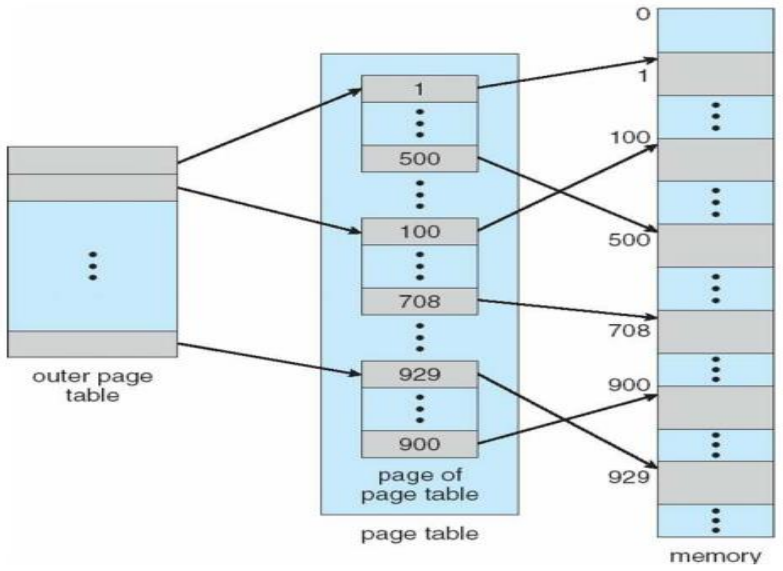
## Hierarchical Paging

- Example (using same values as in previous slide): m − n = 20 can be divided into two 10-bit numbers
- virtual address:



  - th in one 4 KB page
  - w age table, and enough frames to reference the pages actually used by the process

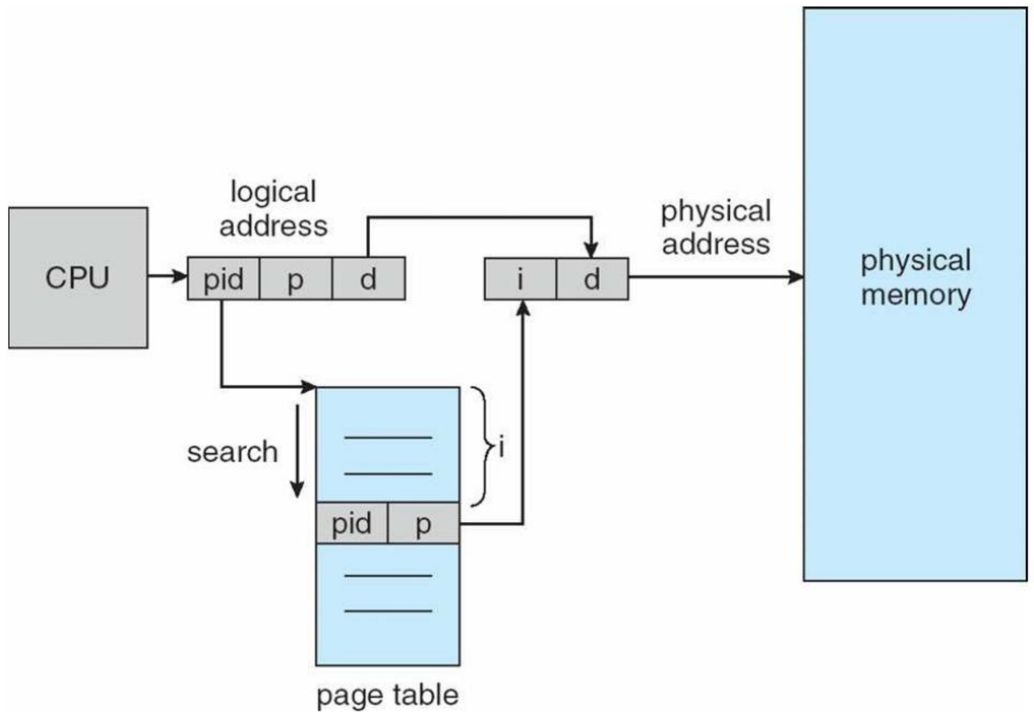
# Hierarchical Paging (Two Levels)



- Drawback: the time to convert a virtual address into a physical one is longer
- This problem is even worse if m is large and we need more than two levels

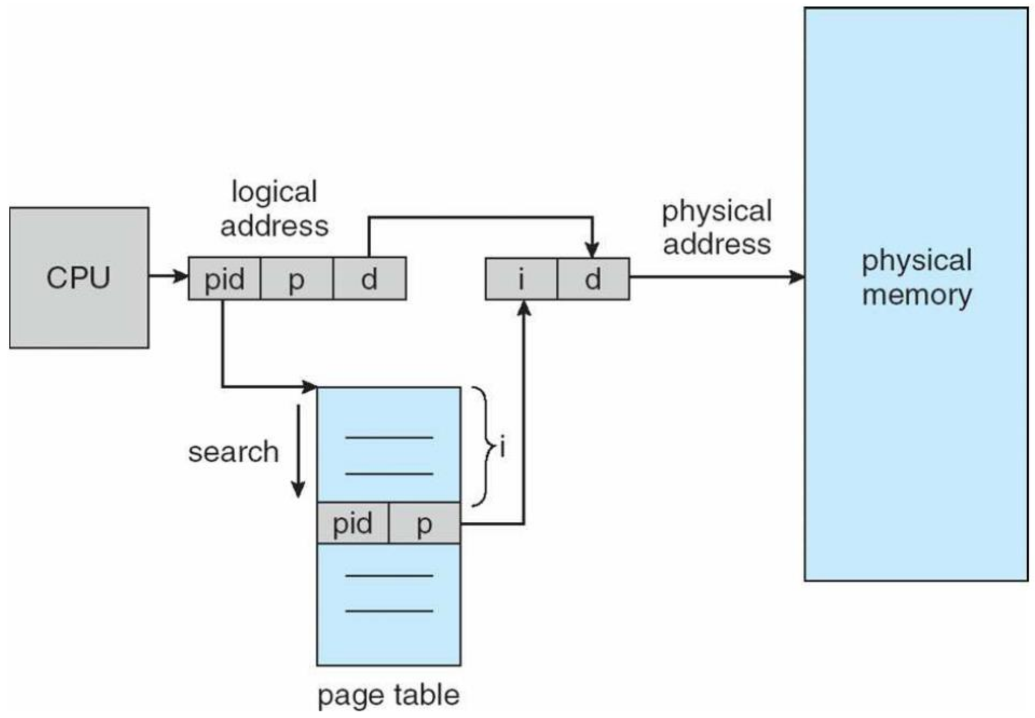
## Inverted Page Table

- For large m a good solution is an inverted page table
  - the inverted page table has one entry per frame of physical memory (so there is only one)
  - each entry includes a PID and page number



# Inverted Page Table

- Issue: sequential search (linear inverted page table)
  - for this reason it is usually hashed to speed up access, using linked lists for collisions
- If no match found: page fault



## Next week

- Remember class time change
- Study Time
  - Review Chapter 9