University of Glasgow | School of Computing Science

Networks & Operating Systems Essentials

Dr Angelos Marnerides

<*angelos.marnerides@glasgow.ac.uk*>

School of Computing Science, Room: S122

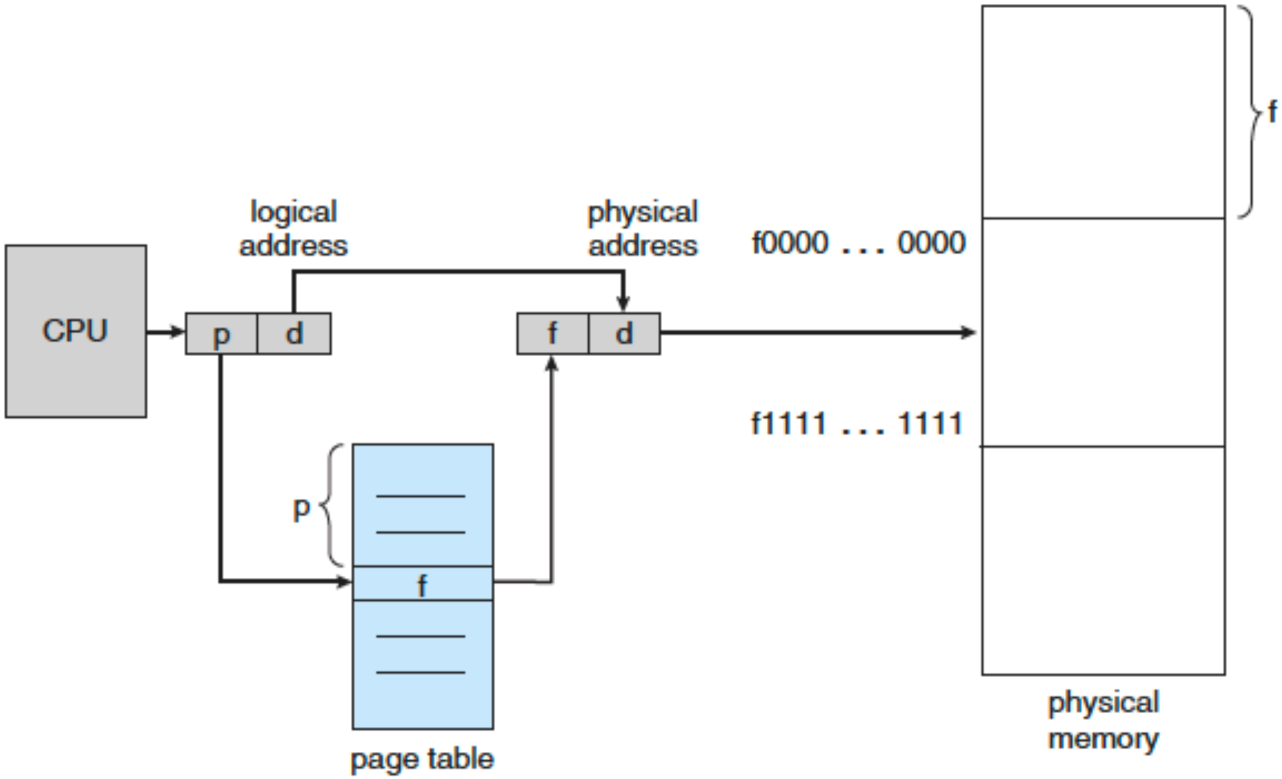
Memory Management (Recap)

- Idea #1→ Let's use special registers to store the first and last address in a process's address space (Base Register & Limit Register)
- Idea #2 → Partition RAM into fixed size partitions, allocate one to each running process
  - Would work, but is inflexible and clunky
  - Creates internal fragmentation: space within partitions goes unused
- Idea #3 → Variable-sized partitions:
  - The OS keeps track of lists of allocated ranges and "holes" in RAM
  - When a new process arrives, it blocks until a hole large enough to fit it is found
  - If hole is too large, it's split in two parts: one allocated to the new process, one added to the list of holes
    - Two or more adjacent holes may be merged
    - The OS may then check whether the newly created hole is large enough for any waiting processes
  - Can create external fragmentation: space in between partitions too small to be used

  - Idea #4:→ Segments : Maintain several "specialised" segments and a table with info for each segment (extension of BR/LR to "mini" address spaces)
    - … but how to map segments to "holes" in RAM?

Paging

Idea #5: Paging



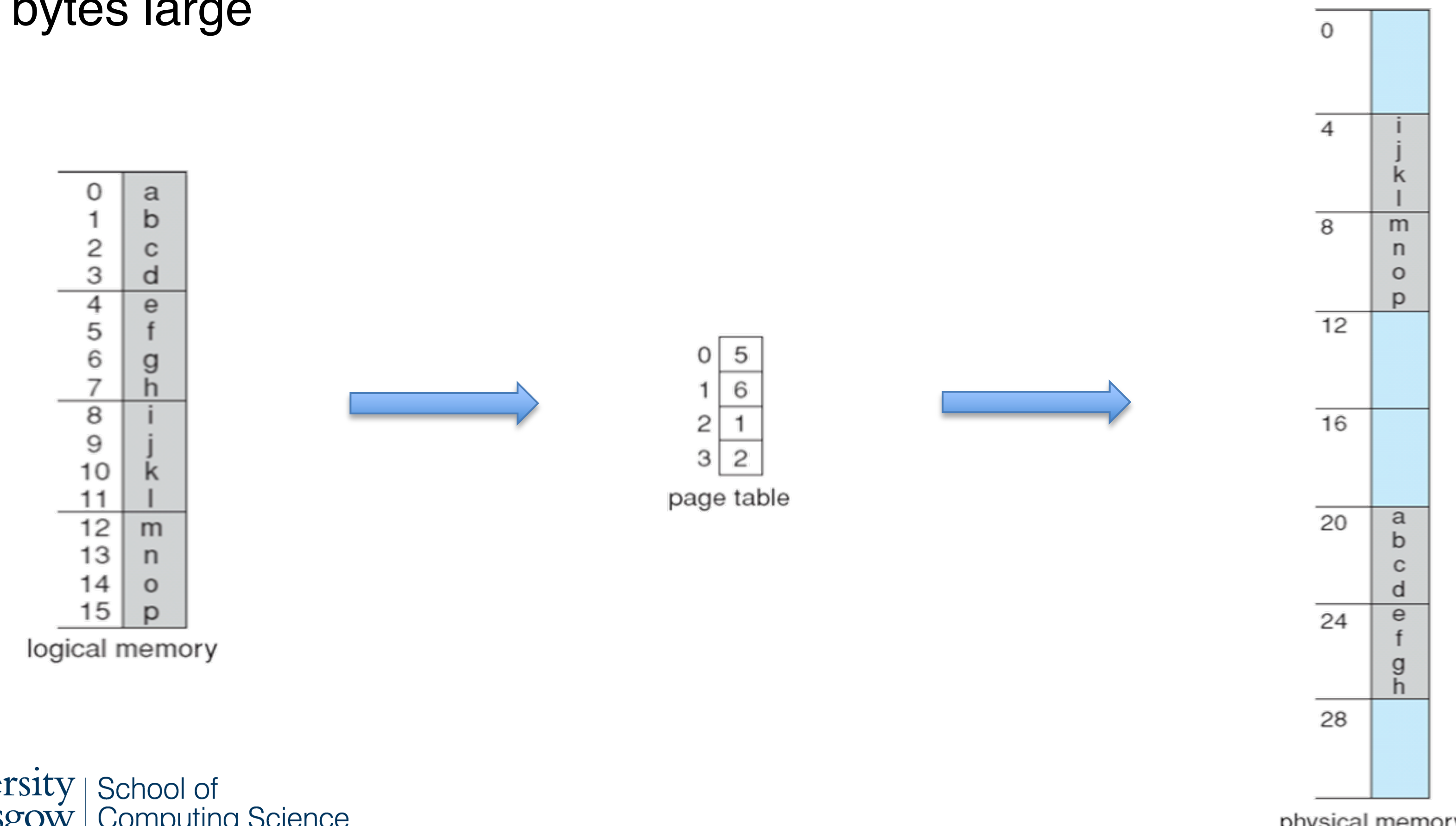
Paging…

- Partition address space in equally sized, fixed-size partitions (page)
  - Size always a power of 2 -- i.e., 2^d
  - Typical page sizes: 1KB – 4KB (could get even bigger in modern OS)
  - Each page is kept on disk, so there can be a lot of them
- A location in the address space can be given as either an *address*, or a *page number* plus an *offset* within set page
- Given an address with n bits:
  - The least significant d bits are the offset
  - The most significant $p = n - d$ bits define the page number

- Partition a large portion of RAM into page frames, each of size equal to a page
- Maintain a page table for each address space; each entry contains:
  - A boolean (resident/valid bit): True if page is loaded into a page frame
  - A frame address: if resident bit is True, contains the physical address of the first location in the frame
  - Can be per process or contain additional data (e.g., process ID) for protection
- Paging means moving a page/frame of data from disk to memory or vice-versa

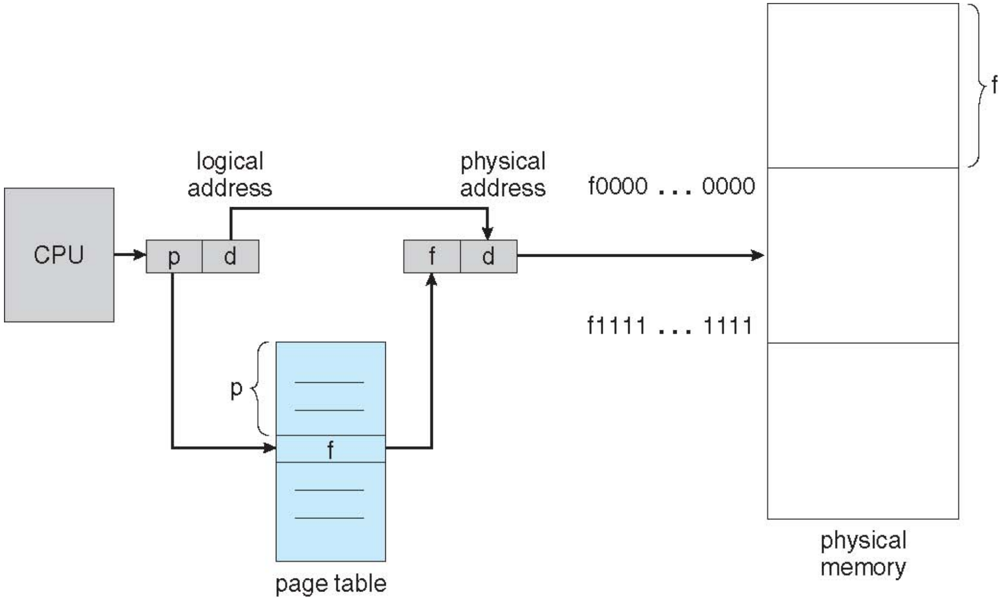
Paging Example

- Assume we have a 32-byte memory -- i.e., n = 5
- Assume 4-byte frames -- i.e., d = 2
- Assume our process's address space (i.e. logical memory) is 16 bytes large



Memory Address Translation

Source: A. Silberschatz, "Operating System Concepts", 9th Ed., 2012.



Wait… What if **p** and **f** were a different size?

Should then **p** be larger or smaller than **f**? p<=f

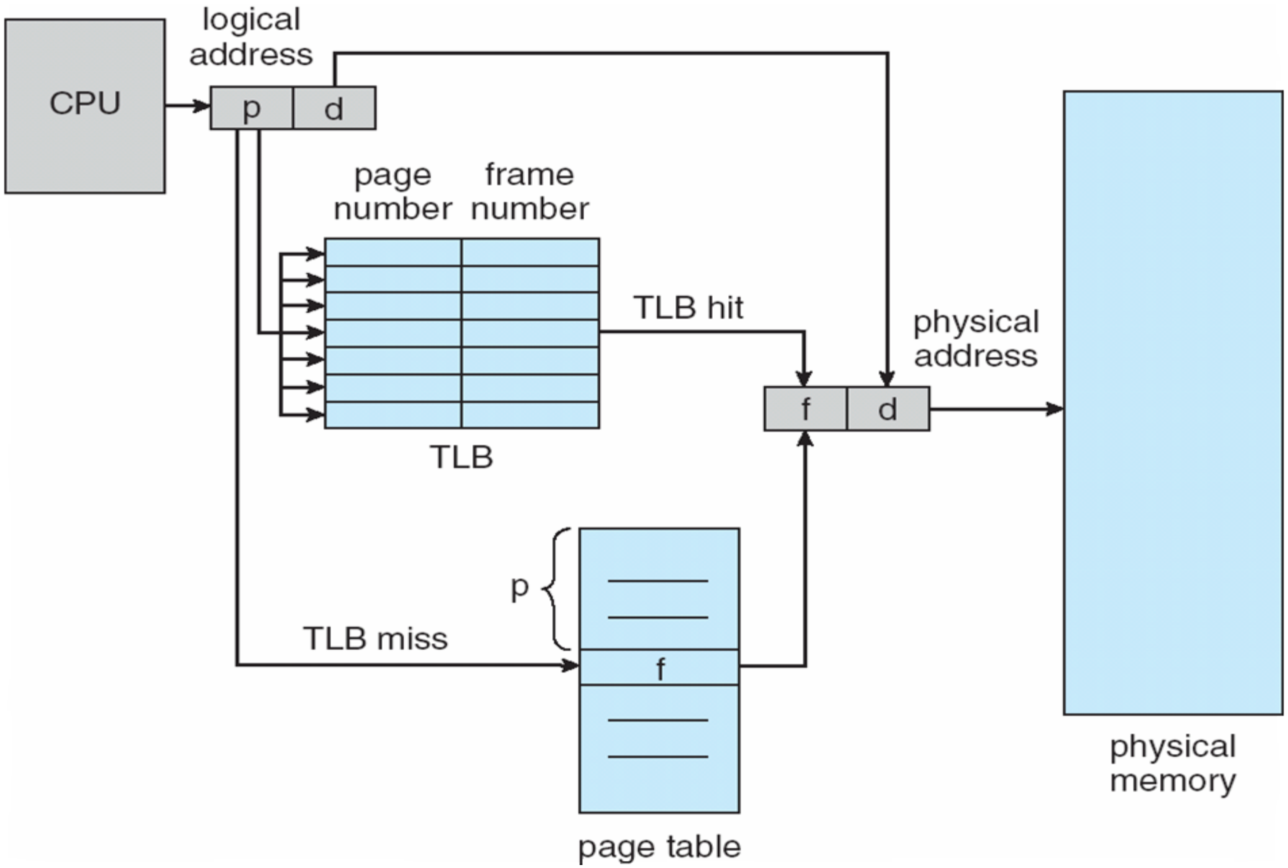
Allocation of Frames

- Each process needs minimum number of frames
  - Maximum is total frames in the system, minus frames allocated to the OS
- Three major allocation schemes:
  - Fixed allocation
    - Divide available frames equally among processes
  - Proportional allocation
    - Give each process a percentage of frames equal to its size divided by the sum of sizes of all processes
  - Priority allocation
    - Like proportional allocation, but taking into account the priority of a process (possibly in combination with its size)
- Speed of access to memory may vary across CPUs -- e.g., NUMA (non-uniform memory access) systems
  - Better allocate memory “close to” the CPU where the process that caused the page fault is running

Implementation considerations

- Page table is an OS construct but with hardware assistance…
- Page table is kept in main memory
  - Page-table base register (PTBR) points to beginning of page table location
  - Page-table length register (PTLR) indicates size of the page table
- But wait! In this scheme **every** data/instruction access requires **two memory accesses**!!!
  - One for the page table and one for the data/instruction
- How is this acceptable?
- Caching to the rescue…
  - Use an on-CPU cache of the Page Table → Translation Lookaside Buffer (TLB)

Translation Lookaside Buffer (TLB)



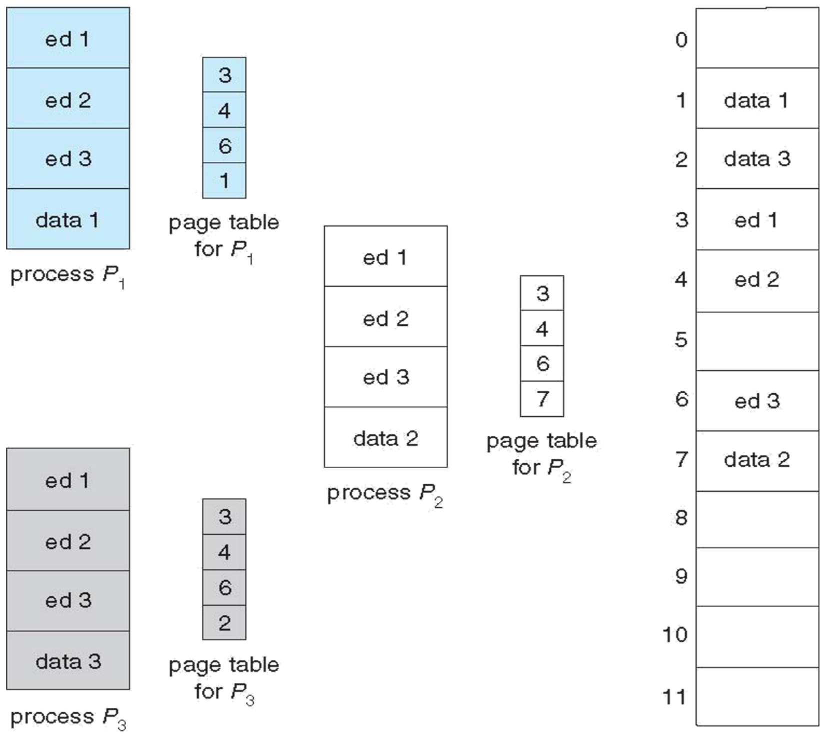
Aside: TLB Effective Access Time

- Consider a system where each memory access takes m = 100ns and each TLB lookup requires t = 10ns
- Now consider a X% TLB hit ratio (ρ)
  - For X% of memory accesses we'll have the physical address from the TLB in 10ns
  - For (100-X)% of memory accesses, we'll need to go to RAM to fetch the mapping to the physical address, taking 10ns + 100ns
  - … plus 100ns for the actual memory access
- Effective Access Time (EAT):
  - $EAT = \rho \times (t + m) + (1 - \rho) \times (t + m + m)$
  - $\rho = 80\% \rightarrow EAT = 0.8 \times (10 + 100) + 0.2 \times (10 + 100 + 100) = 130ns$
  - $\rho = 99\% \rightarrow EAT = 0.99 \times (10 + 100) + 0.01 \times (10 + 100 + 100) = 111ns$
- Not that bad after all…

Shared Pages

- Pages may be shared across processes
  - Often done for pages containing code
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers)

- Remember *fork*?
  - Expensive as should create fully copy of parent process's address space
- Can we make it any faster?
- Yes, with Copy-on-Write (COW)
  - Both processes share the same pages in memory
  - If a process tries to modify a page, a private copy is created



Implementation considerations (cont.)

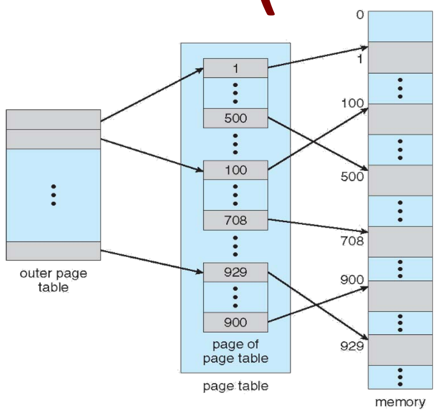
- Time for some quick math…
  - 4 kB page frames in a 64-bit address space
  - 4kB = 2^{12} bytes ➔ 12-bit offset ➔ 52-bit page no
  - ➢ 2^{52} pages in the page table
  - Assume 2×64 bits (=16 bytes) per entry of the page table
  - ➢ $16 \times 2^{52} = 2^{56}$ = **32 petabytes!**

- Now the page table **cannot fit in the RAM**!!!

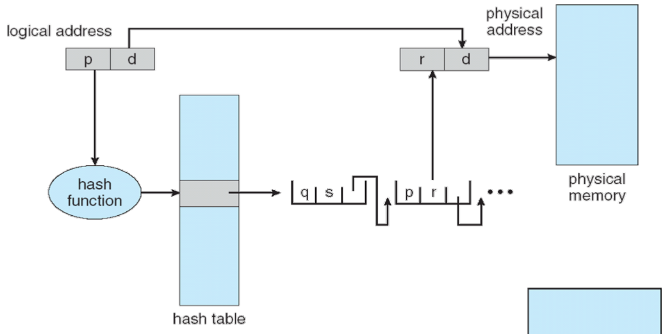
- Solution: partition the page table…

Implementation considerations (cont.)

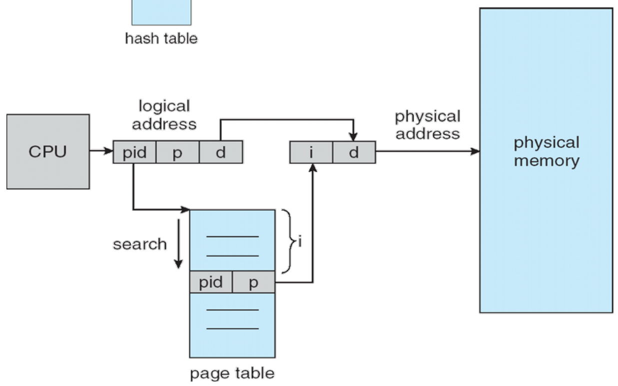
- Idea #1: Hierarchical page tables
  - 2-level PTs quite common



- Idea #2: Hashed page tables
  - Hash table of linked lists of mappings
  - Can contain multiple mappings per entry (clustered PTs)

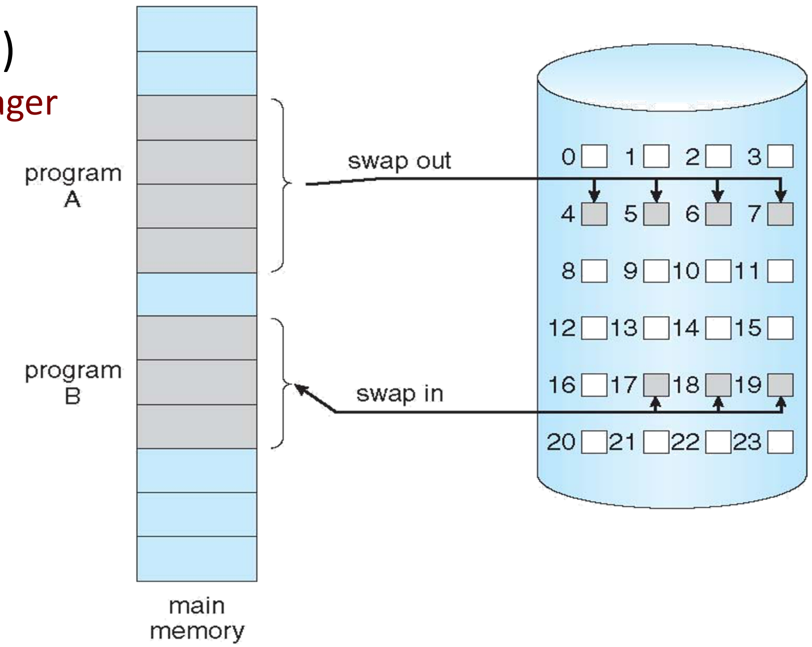


- Idea #3: Inverted page table
  - Maintain mappings of each physical frame to virtual addresses
  - Mapping must also contain owner process id



Paging revisited

- Remember:
  - Each page is kept on disk, so there can be a lot of them
  - Paging means moving a page/frame of data from disk to memory or vice-versa
  - Originally, all pages are on disk (e.g., before executing a program)
- When should a page be moved from disk to memory?
  - All pages on process startup
  - Every page only as needed (accessed)
    - **Demand paging** -- i.e., using a “lazy” pager



Demand Paging

- Remember the “resident/valid bit”?
- What if a process tries to access a page that isn’t loaded in RAM?
  - **Page fault!**
- The pager (a.k.a. page fault handler) kicks in…
  1. Trap to OS (process is blocked → context switch)
  2. Kernel computes page location on disk
  3. Kernel issues a disk read to load contents of the page to a ***free frame***
  4. Jump to the process scheduler (other processes execute)
  5. Interrupt from the disk (I/O completed)
  6. Context switch & control passed to pager
  7. Pager updates the page table (frame address, resident bit)
  8. Process moved to the READY queue
  9. Pager returns to process scheduler

Demand Paging

- Virtual memory can run almost as fast as real memory, **if the proportion of instructions that cause a page fault is low enough**
- In the real world, programs perform almost all their memory accesses at addresses close to where they recently accessed data
  - Nearly all accesses are to a **working set** which is kept in page frames
- Page faults occur when a program changes its working set (e.g. when a method has finished, and calls another)
- Occasionally, a program will perform too many page faults
  - The result is that it starts to execute thousands of times slower than it should
  - This is called **thrashing**

Demand Paging

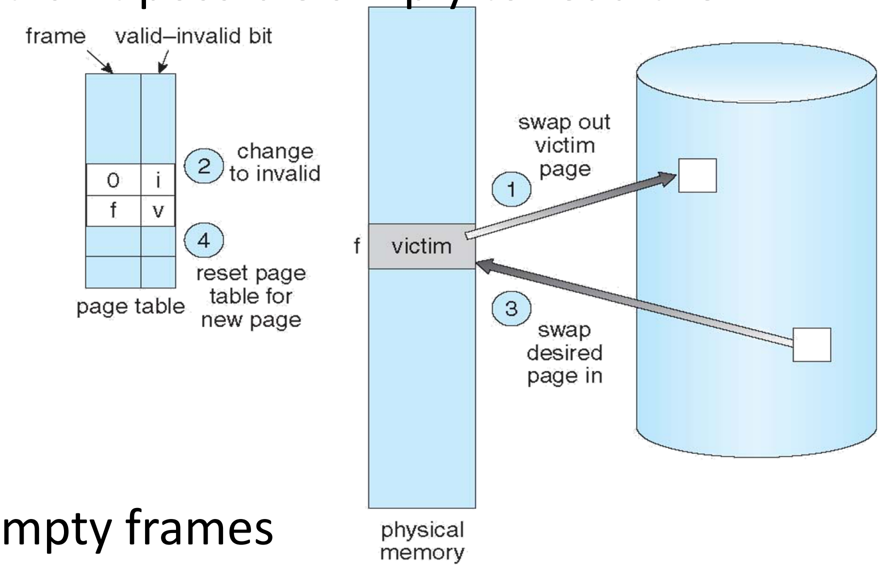
- Time for some more quick math…
- Suppose you execute 100 instructions; 1 causes a page fault and 99 don't
- How much slower will this run than if there were no page fault?
  - Typical time to execute an instruction: 1ns (if cache hit)
  - Typical disk access time: 50ms
- We'll *estimate* this, not calculate it exactly
  - Time for 100 instructions (no page fault) = 1ns / instruction $\times$ 100 instructions = 100 ns
  - Time for 99 instructions = 99 ns ~= 10^2 ns
  - Time for 1 page fault *assuming free frame exists* ~= 50 ms = 50×10^6 ns = 5×10^7 ns
  - Slowdown = $(5 \times 10^7)/10^2$ = 500,000
- Conclusion: if you have as many as 1% page faults, your program will run half a million times slower!
- Generally, if page fault rate: $p \in [0, 1]$, then Effective Access Time (EAT) =

  $(1 - p) \times$ memory access +

  $p \times$ (page fault overhead + swap page out + swap page in + restart overhead)

Page Replacement

- Normally, most or all of the page frames are in use
- In this case, when a page fault occurs, it isn't possible simply to read the page into an empty frame



- The system must:
  1. Select a full frame (**page replacement strategy/algorithm**)
  2. Write its contents to disk (page-out)
  3. Then read the required page into this frame (page-in)

- In practice, it's better to keep several empty frames
  - On a page fault, the read (to load the data) can start immediately
  - A separate write (to clear another frame) can be done right after that (or *lazily* even later)
- Can further optimise page-outs by introducing "dirty/modified" bit
  - Set to 1 if page has been updated → only write page to disk if true

Page Replacement

- Note: With page replacement, larger virtual memory can be provided on a smaller physical memory

- Page replacement requires hardware…
  - Circuitry in the CPU must:
    1. Find the page number
    2. Look up the page table entry
    3. Check for residency
    4. If resident, find the real memory address
    5. If not resident, generate an interrupt
  - Has to be done in hardware, for efficiency
- ... and software
  - In the event of a page fault, the pager must:
    - Perform the disk I/O
    - Maintain the page table
    - Implement the page replacement policy
  - Has to be done in software, because of complexity and flexibility

Page Replacement

- If a page is written from a frame back to disk, and data on this page is needed soon, it will have to be read back in
  - This results in too much disk I/O (**slow**)
- Therefore, the pager doesn't randomly choose a frame to clear
  - It has a sophisticated page replacement policy
  - Goal: Try to choose a frame containing a page that probably won't be needed again soon
- Enter page replacement algorithms…
- Alternatives:
  - Global replacement: consider all (non-OS) frames
  - Local replacement: only consider frames allocated to the process that caused the page fault

Page Replacement Algorithms

- Optimal (OPT or MIN):
  - Page-out the page that won't be used for the longest period of time -- unrealistic
- First-In-First-Out (FIFO):
  - Store the page-in time of every page; choose oldest page to page-out
  - Add loaded pages at the tail of a queue; choose the head of the queue to page out
- Least Recently Used (LRU):
  - Store the time of use of every page; choose the page that hasn't been used for the longest
  - Move accessed pages at the tail of a queue; choose the head of the queue to page-out
- Random:
  - Pick a page at random
  - Generally better than FIFO but worse than LRU in practice

Page Replacement Algorithms (cont.)

- Simple reference-bit-based:
  - Maintain an “accessed bit” per page (initially 0); choose first page with a 0 bit
- Least Frequently Used (LFU) / Non Frequently Used (NFU)
  - Maintain counter of accesses to each page; choose the page with the lowest count to page-out
- Aging / additional reference bits:
  - Maintain a multi-bit word per page; set MSB to 1 on access; shift right regularly; choose page with lowest value to page-out
- Second-chance (clock):
  - Maintain an “accessed bit” per page; run FIFO to select a page; if accessed bit is 0, page-out; otherwise (if 1), set to 0, move the page at the tail of the queue and re-run FIFO

Page Replacement Algorithms (cont.)

- Not Recently Used (NRU):
  - Maintain "accessed" and "modified" bits per page; use the two bits (accessed, modified) as a 2-bit score; choose the first page with the lowest score
    - (0,0) = 0: not accessed, not modified (best candidate) ➔ if found, page-out
    - (0,1) = 1: not recently used but modified (will need write out to disk) ➔ write-out, clear modified bit, continue the search
    - (1,0) = 2: recently used but not modified (better keep it as might be used again soon) ➔ clear accessed bit, continue the search
    - (1,1) = 3: accessed and modified (worst candidate) ➔ clear accessed bit, continue the search
    - Multiple passes may be required; by the 3rd pass, all pages will be at (0,0)

Page Replacement Algorithm Evaluation

- Evaluation methodology:
  - Use fixed sequence of page accesses
  - Evaluated by computing the number of page faults (denoted by *)
- Running example:
  - Consider a cache with 3 slots and the following stream of requests: A, B, C, A, B, B, D, A, C, D, B
    - A, B, C, D: page numbers/addresses

OPT example

- Access string: B, C, A, B, B, D, A, C, D, B
- '*' = page fault

B	C	A	B	B	D	A	C	D	B
B	B	B	B	B	D	D	D	D	D
	C	C	C	C	C	C	C	C	B
		A	A	A	A	A	A	A	A
*	*	*			*				*

Frames

RANDOM example

- Access string: B, C, A, B, B, D, A, C, D, B
- '*' page fault

B	C	A	B	B	D	A	C	D	B
B	B	B	B	B	B	B	C	C	C
	C	C	C	C	D	D	D	D	B
		A	A	A	A	A	A	A	A
*	*	*			*		*		*

Frames

FIFO example

- Access string: B, C, A, B, B, D, A, C, D, B
- Maintaining page-in time for every page
- '*' = page fault

B	C	A	B	B	D	A	C	D	B
B (0)	B (0)	B (0)	B (0)	B (0)	D (5)	D (5)	D (5)	D (5)	D (5)
	C (1)	C (1)	C (1)	C (1)	C (1)	C (1)	C (1)	C (1)	B (9)
		A (2)	A (2)	A (2)	A (2)	A (2)	A (2)	A (2)	A (2)
*	*	*			*				*

Frames

LRU example

- Access string: B, C, A, B, B, D, A, C, D, B
- Maintaining last access time for every page
- ‘*’ = page fault

B	C	A	B	B	D	A	C	D	B
B (0)	B (0)	B (0)	B (3)	B (4)	B (4)	B (4)	C (7)	C (7)	C (7)
	C (1)	C (1)	C (1)	C (1)	D (5)	D (5)	D (5)	D (8)	D (8)
		A (2)	A (2)	A (2)	A (2)	A (6)	A (6)	A (6)	B (9)
*	*	*			*		*		*

Frames

LFU example

- Access string: B, C, A, B, B, D, A, C, D, B
- Maintaining access counts for every page
- ‘*’ = page fault

B	C	A	B	B	D	A	C	D	B
B (1)	B (1)	B (1)	B (2)	B (3)	B (3)	B (3)	B (3)	B (3)	B (4)
	C (1)	C (1)	C (1)	C (1)	D (1)	D (1)	C (1)	D (1)	D (1)
		A (1)	A (1)	A (1)	A (1)	A (2)	A (2)	A (2)	A (2)
*	*	*			*		*	*	

Frames

Reference Bit example

- Access string: B, C, A, B, B, D, A, C, D, B
- Maintaining reference bit for every page, resetting after two accesses
- '*' = page fault

B	C	A	B	B	D	A	C	D	B
B (1)	B (1)	B (0)	B (1)	B (1)	B (1)	B (0)	C (1)	C (1)	C (0)
	C (1)	C (1)	C (0)	C (0)	D (1)	D (1)	D (0)	D (1)	D (1)
		A (1)	A (1)	A (0)	A (0)	A (1)	A (1)	A (0)	B (1)
*	*	*			*		*		*

Frames

Aging example

- Access string: B, C, A, B, B, D, A, C, D, B
- Maintaining 3 bits for every page, shifting on every access
- ‘*’ = page fault

B	C	A	B	B	D	A	C	D	B
B (100)	B (010)	B (001)	B (100)	B (110)	B (011)	B (001)	C (100)	C (010)	C (001)
	C (100)	C (010)	C (001)	C (000)	D (100)	D (010)	D (001)	D (100)	D (010)
		A (100)	A (010)	A (001)	A (000)	A (100)	A (010)	A (001)	B (100)
*	*	*			*		*		*

Frames

Second-chance (clock) example

- Access string: B, C, A, B, B, D, A, C, D, B
- Maintaining page-in time and access bit for every page → (page_in, access)
- '*' = page fault

B	C	A	B	B	D	A	C	D	B
B (0,1)	B (0,1)	B (0,1)	B (0,1)	B (0,1)	D (5,1)	D (5,1)	D (5,1)	D (5,1)	D (5,0)
	C (1,1)	C (1,1)	C (1,1)	C (1,1)	C (1,0)	C (1,0)	C (1,1)	C (1,1)	B (9,1)
		A (2,1)	A (2,1)	A (2,1)	A (2,0)	A (2,1)	A (2,1)	A (2,1)	A (2,0)
*	*	*			*				*

Frames

NRU example

- Access string: B, C, A, <u>B</u>, B, D, <u>A</u>, C, <u>D</u>, B
- Maintaining page-in time, and access and modify bit for every page → (page_in, access, modify)
- Assume underlined accesses are modifications
- '*' = page fault

B	C	A	<u>B</u>	B	D	<u>A</u>	C	<u>D</u>	B
B (0,1,0)	B (0,1,0)	B (0,1,0)	B (0,1,1)	B (0,1,1)	B (0,0,1)	B (0,0,1)	C (7,1,0)	C (7,1,0)	C (7,0,0)
	C (1,1,0)	C (1,1,0)	C (1,1,0)	C (1,1,0)	D (5,1,0)	D (5,1,0)	D (5,0,0)	D (5,1,1)	D (5,0,1)
		A (2,1,0)	A (2,1,0)	A (2,1,0)	A (2,0,0)	A (2,1,1)	A (2,0,1)	A (2,0,1)	B (9,1,0)
*	*	*			*		*		*

Recommended Reading

- Silberschatz, Galvn and Gagne, *Operating Systems Essentials, Chapter 7, sections 7.3,7.4, 7.5*