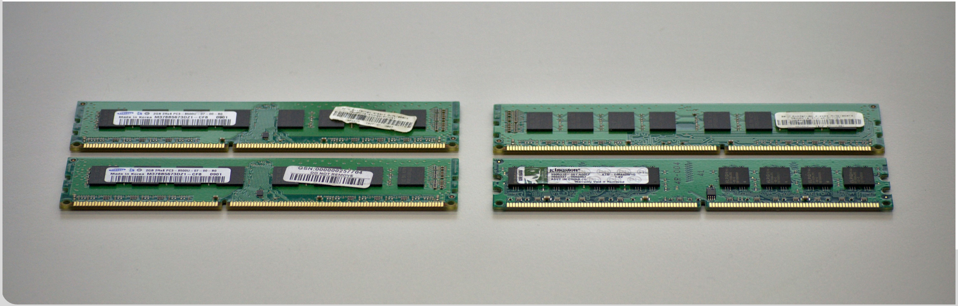
Operating Systems

08. Page Fault Handling

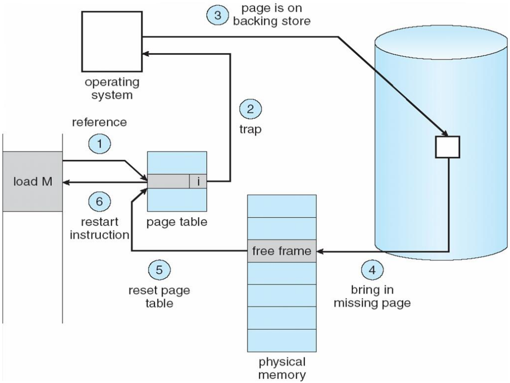
Prof. Dr. Frank Bellosa | WT 2020/2021



Page Faults

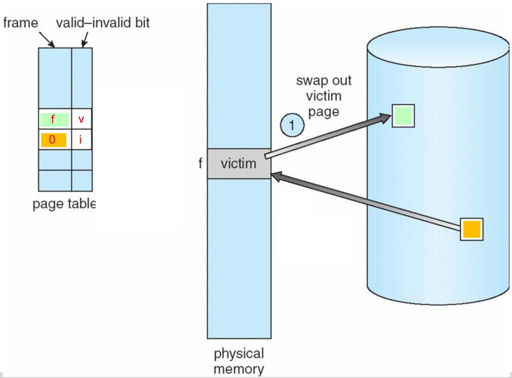
Page Fault Handling [KELS62]

- Access to page that is currently not present in main memory causes page fault (exception that invokes OS)
  - OS checks validity of access (requires additional info)
  - Get empty frame (free or victim)
  - Prepare requested page (e.g., clean or load content from disk into frame)
  - Adapt page table
  - Set present bit of respective entry (a.k.a. as setting valid-invalid bit to **v**)
  - Restart instruction that caused the page fault [SGG12]



Page Replacement [SGG12]

- Save/clear victim page (see chapter on page replacement policies)
  - Drop page if fetched from disk (e.g., code) and clean (PTE dirty bit)
  - Write back modifications if from disk and dirty
  - Write pagefile/swap partition otherwise (e.g., stack, heap memory)
- Unmap page from old AS
  - unset valid bit in PTE
  - flush TLB
- Prepare the new page
  - e.g., NULL page
  - e.g., load new contents
- Map the page frame into the new address space(s)
  - set valid bit in PTE
  - flush TLB



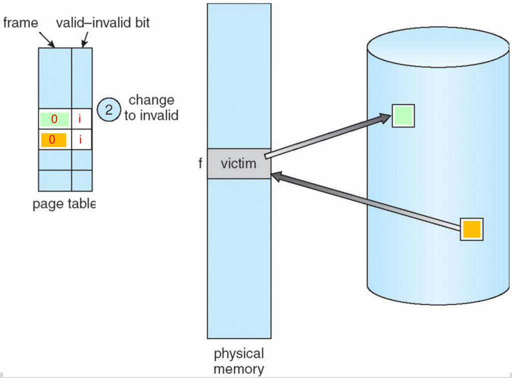
Page Replacement [SGG12]

- Save/clear victim page (see chapter on page replacement policies)
  - Drop page if fetched from disk (e.g., code) and clean (PTE dirty bit)
  - Write back modifications if from disk and dirty
  - Write pagefile/swap partition otherwise (e.g., stack, heap memory)

- Unmap page from old AS
  - unset valid bit in PTE
  - flush TLB

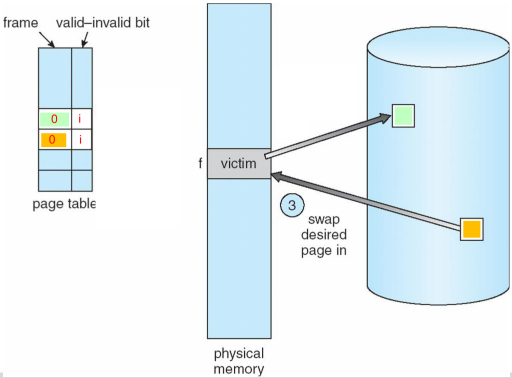
- Prepare the new page
  - e.g., NULL page
  - e.g., load new contents

- Map the page frame into the new address space(s)
  - set valid bit in PTE
  - flush TLB



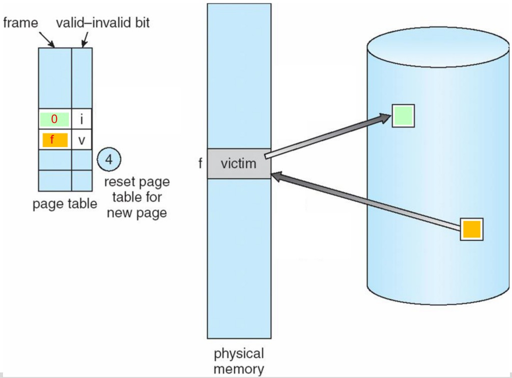
Page Replacement [SGG12]

- Save/clear victim page (see chapter on page replacement policies)
  - Drop page if fetched from disk (e.g., code) and clean (PTE dirty bit)
  - Write back modifications if from disk and dirty
  - Write pagefile/swap partition otherwise (e.g., stack, heap memory)
- Unmap page from old AS
  - unset valid bit in PTE
  - flush TLB
- Prepare the new page
  - e.g., NULL page
  - e.g., load new contents
- Map the page frame into the new address space(s)
  - set valid bit in PTE
  - flush TLB



Page Replacement [SGG12]

- Save/clear victim page (see chapter on page replacement policies)
  - Drop page if fetched from disk (e.g., code) and clean (PTE dirty bit)
  - Write back modifications if from disk and dirty
  - Write pagefile/swap partition otherwise (e.g., stack, heap memory)
- Unmap page from old AS
  - unset valid bit in PTE
  - flush TLB
- Prepare the new page
  - e.g., NULL page
  - e.g., load new contents
- Map the page frame into the new address space(s)
  - set valid bit in PTE
  - flush TLB



Page Fault Latency

- Page Fault Rate $0 \leq p \leq 1.0$
  - $p = 0$: No page faults
  - $p = 1$: Every reference is a page fault
- Effective Access Time (EAT)

$$
\begin{aligned}
\text{EAT} \quad = \quad (1-p) \quad & \times \quad \text{memory access} \\
+ \quad p \times & \Big(\quad \text{page fault overhead} \\
& \quad + \text{page fault service time} \\
& \quad + \text{restart overhead} \quad \Big)
\end{aligned}
$$

Performance Impact of Page Faults

- Memory access time = 200 nanoseconds
- Average page fault service time = 8 milliseconds
- $$\begin{aligned} \text{EAT} &= (1-p)\times 200 + p(8\text{ms}) \\ &= (1-p)\times 200 + p\times 8,000,000 \\ &= 200 + p\times 7,999,800 \end{aligned}$$
- If one access out of 1,000 causes a page fault, then EAT = 8.2 microseconds. $\Rightarrow$ Slowdown by a factor of 40!

What to fetch?

- Bring in page that caused page fault
- Pre-fetch surrounding pages?
  - Reading two disk blocks is approximately as fast as reading one
  - As long as no track/head switch, seek time dominates (disk)
  - If application exhibits spatial locality → big win
- Pre-zero pages?
  - Don't want to leak information between processes
  - Need 0-filled pages (0-pages) for stack, heap, .bss, . . .
  - Zero on demand?
  - Keep a pool of 0-pages that is filled in the background when the CPU is idle?

How to resume a process after a fault?

- Hardware provides info about page fault
  - Faulting virtual address: **`%cr2`** on intel

- OS needs to figure out context of fault. Was the instruction a
  - Read or write?
  - Instruction fetch?
  - User access to kernel memory?

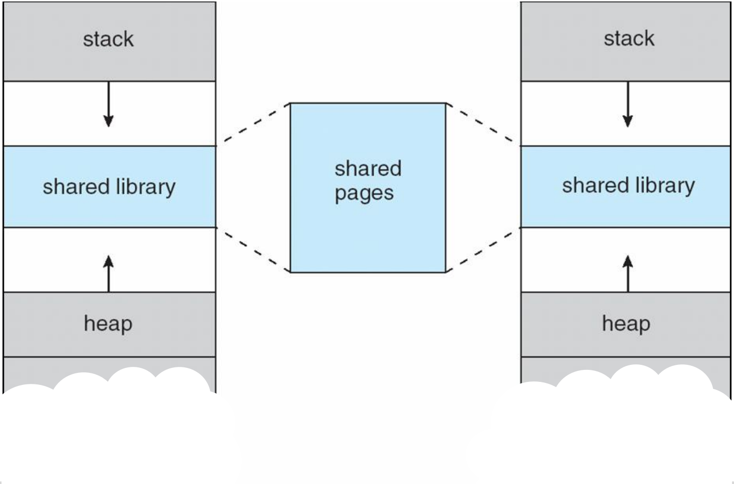
- Idempotent instructions are easy
  - Just re-do load/store instructions
  - Just re-execute instructions that only access one address

Complex instructions must be re-started, too

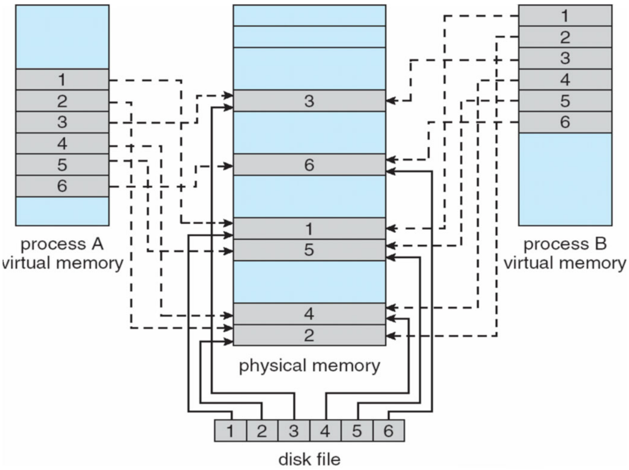
- Some CISC instructions are difficult to restart such as
  - Block move of overlapping areas, string move instructions
  - Auto-increments/decrements of multiple locations
  - Instructions that keep and update state in source **`%esi`**, destination **`%edi`**, and counter **`%ecx`** registers

- Possible Solutions
  - Touch all relevant pages before operation starts
  - Keep modified data in registers so that page faults can't take place
  - Design ISA such that complex operations can execute partially and leave consistent state on a page fault (easy job for the OS)

Shared Library/Code Using Virtual Memory [SGG12]



Memory-Mapped Files [SGG12]

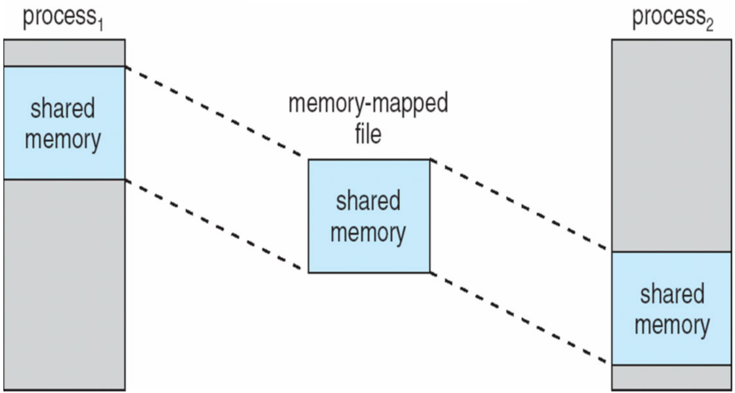


Other Issues – Memory-Mapped Files

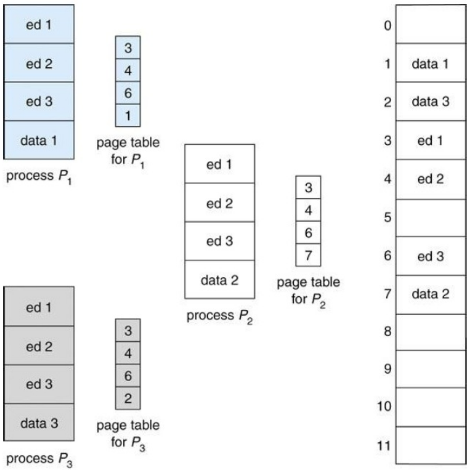
- Memory-mapped file I/O allows file I/O to be treated as routine memory access by **mapping** a disk block to a page in memory
- A file is initially read using demand paging. A page-sized portion of the file is read from the file system into a physical page. Subsequent reads/writes to/from the file are treated as ordinary memory accesses
- Simplifies file access by treating file I/O through memory rather than `read()` `write()` system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared

Shared Data Segments [SGG12]

- Shared data segements are often implemented with
  - temporary, anonymous memory-mapped files
  - shared pages (with allocated space on backing store)



Shared Pages Example [SGG12]

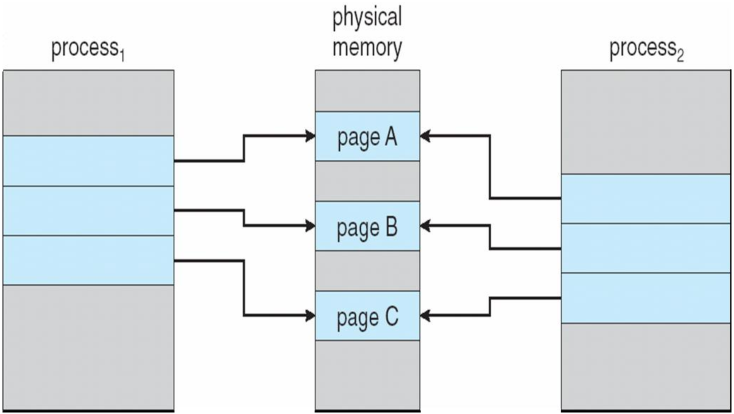


Copy-On-Write

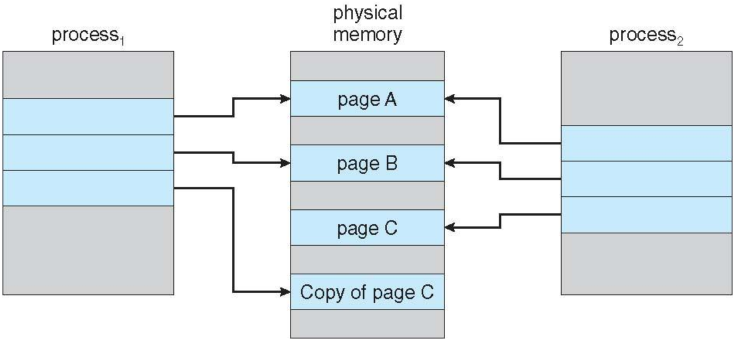
Copy-on-Write (COW) allows multiple processes
to initially share the same pages in memory

- A page is copied only if one of the processes attempts to modify it
- COW allows more efficient process creation as only modified pages are copied

COW: Before Process 1 Modifies Page C [SGG12]



COW: After Process 1 Modifies Page C [SGG12]



Frame Allocation

Local vs. Global Allocation

- Global allocation: All frames are considered for replacement
  - Does not consider page ownership
  - One process can get another process's frame
  - Does not protect process from a process that hogs all memory

- Local allocation: Only frames of the faulting process are considered for replacement
  - Isolates processes (or users)
  - Separately determine how many frames each process gets

Fixed Allocation

- Equal allocation: All processes get the same amount of frames
  - e.g., there are 100 frames and 5 processes → each process gets 20 frames

- Proportional allocation: Allocate according to the size of the process

$s_i =$ size of process p_i
$S = \sum s_i$
$m =$ total number of frames

a_i is the allocation for p_i: $a_i = \frac{s_i}{S} \times m$

Example:

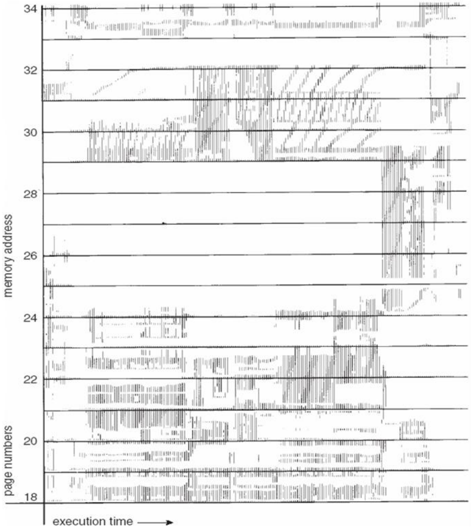
$m = 64$
$s_1 = 10$
$s_2 = 127$

$a_1 = \frac{10}{137} \times 64 \approx 5$
$a_2 = \frac{127}{137} \times 64 \approx 59$

Priority Allocation

- Priority Allocation (global replacement)
  - Proportional allocation scheme using priorities rather than size
- If process P_i generates a page fault
  - Select one of its frames for replacement or
  - Select a frame from a process with lower priority

Locality in a Memory-Reference Pattern [SGG12]



Memory Locality

- Background storage is much slower than memory
  - Paging extends memory size using background storage
  - Goal: Run near memory speed, not near background storage speed

- Pareto principle applies to working sets of processes
  - 10% of memory gets 90% of the references
  - Goal: Keep those 10% in memory, the rest on disk

- Problem: How do we identify those 10%?

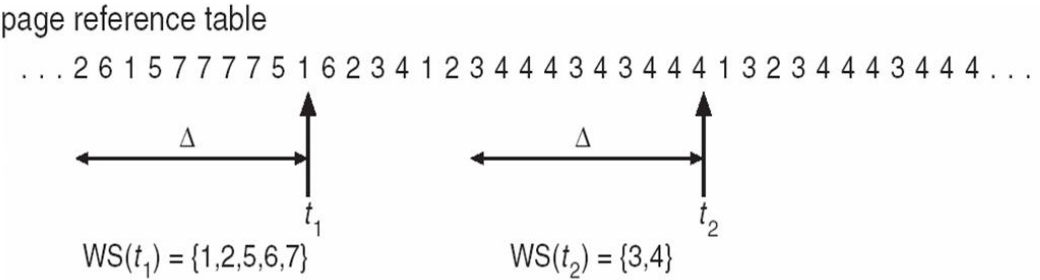
Working-Set Model

- Δ: Working-set window
  - A fixed number of page references
  - Example: 10,000 instructions ($\#$instruction $=$ $\#$page ref?)

- WSS_i: Working set of process P_i
  - Total number of pages referenced in the most recent Δ (varies in time)
  - Δ too small: Will not encompass entire locality
  - Δ too large: Will encompass several localities
  - $\Delta = \infty$: Will encompass entire program

- $D = \sum \text{WSS}_i$: Total demand for frames
- If $D > m$: Thrashing
  - Policy: If $D > m$, suspend a process

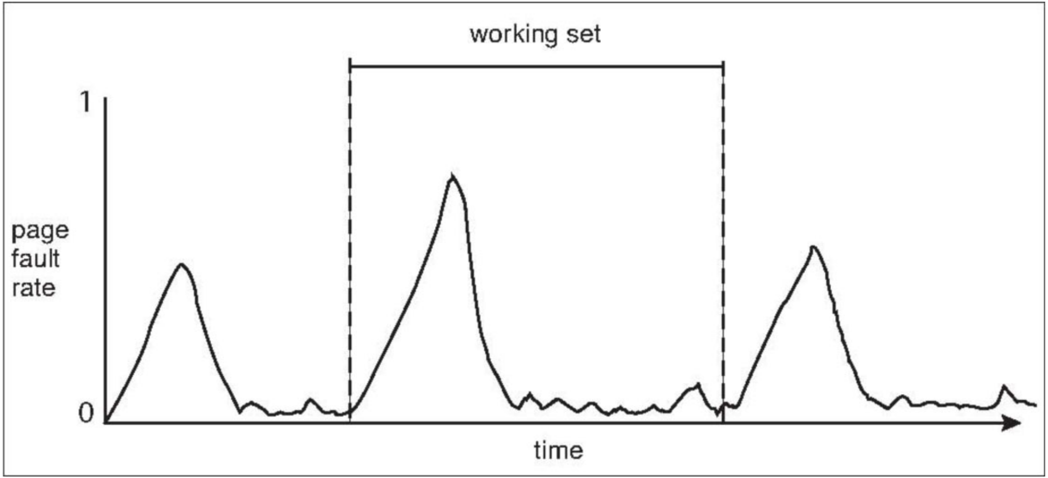
Working-Set Model [SGG12]



Example: Keeping Track of the Working Set

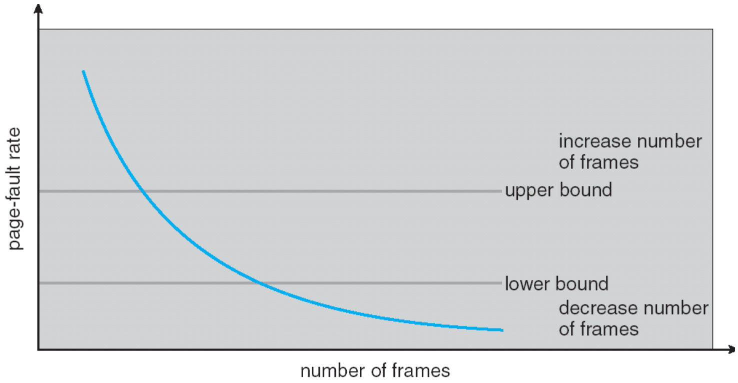
- The MMU automatically sets the reference bit in the respective page table entry every time a page is referenced
- Set a timer to scan all page table entries for reference bits
  - e.g., $\Delta = 10,000$
  - Timer interrupts after every 5,000 time units
  - Keep 2-bit history for each page in addition to the reference-bit
    On timer interrupt, do for each page:
    - Shift reference bit into the 2-bit history
    - Reset reference bit
  - If history $\neq$ 0: Page is in working set
  - Not accurate, because window is moving in large steps
    - Improvement: 10 bits and interrupt every 1000 time units

Working Set and Page Fault Rates [SGG12]



Page Fault Frequency Allocation Scheme

- Establish “acceptable” page fault rate
  - If actual rate too low, give frames to other process
  - If actual rate too high, allocate more frames to process [SGG12]



Page Fetch Policy: Demand-Paging

- When should the OS allocate new pages?
  - Two possibilities: Pre-paging and demand-paging

- Demand-Paging: Transfer only pages that raise page faults

+ Only transfer what is needed
+ Less memory needed per process
  (higher degree of multiprogramming possible)

– “Many” initial page faults when a task starts
– More I/O operations → More I/O overhead

Page Fetch Policy: Pre-Paging

- Pre-Paging: Speculatively transfer pages to RAM
  - At every page fault: speculate what else should be loaded
  - E.g., load entire text section when starting process

+ Improves disk I/O throughput by reading chunks

– Wastes I/O bandwidth if page is never used

– Can destroy the working set of other processes in case of page stealing

References I

[KELS62] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. One-level storage system. *IRE Transactions on Electronic Computers*, EC-11(2):223–235, 1962.

[SGG12] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 9th edition, 2012.