Memory Implementation

Module 13

Theory vs. Practice

Most of what we’ve discussed is “theory” - general ideas.

When implementing, there are some details that we must begin to worry about… and they do matter!

Page Allocation: Local or Global

- We’ve explained that ***working set*** theory tells us that there should be an *optimal* number X which represents the number of pages “active”.

- In practice - on a page fault we have two choices:
  1. Replace one of the pages in the process (local),
  2. Steal a frame from another (global)?

Problems with Local Replacement

Inherently assumes X (from working set theory) is **unchanged**. This assumption might not be valid!

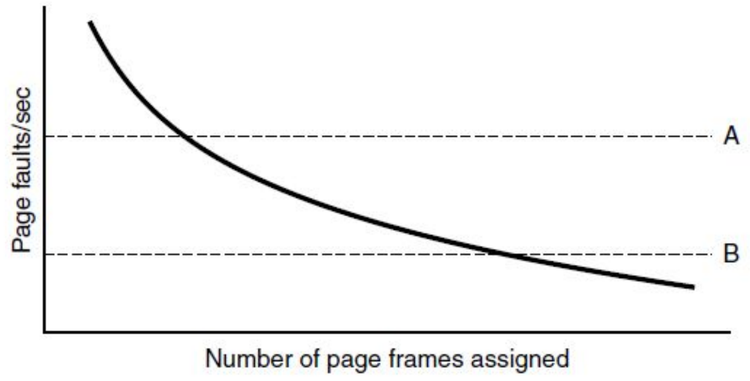
- Could result in thrashing
- Could result in over-allocation (wasted memory)

Benefits of Global Allocation

- We may choose to replace the oldest unreferenced page, globally…
- If a page is really old, it probably isn’t in the working set any longer - we effectively reduce “X” for that process.
    - Dynamically approximates working set.
    - But… what if we are wrong?

Global Allocation - not enough

- We still need to monitor working set:
- If too many processes have large working sets, we begin to **thrash**.



Processes who are thrashing must be suspended (put on backing store)

Suspending Thrashing Processes

- We prefer to remove ***all*** pages from memory when dealing with a thrashing program:
  a. We don't have enough frames available to avoid page faults
  b. Freeing up all frames allows us to more quickly process other tasks
- Of course, we can go too far…
  a. Must maintain enough processes in memory to achieve multiprogramming.
  b. Must implement a "fair" two-tier scheduler.

Page Sizes

We know that page size is chosen by OS.

- Pages are multiples of the size the hardware is design for:
  - If hardware is designed for 4096 byte transfers, pages could be 4KB, 8KB, 16KB…
  - How do we choose page size?

Arguments for Small Pages

- Small pages ultimately reduce internal fragmentation
- Moreover, small programs (or small working sets) are always wasting space!

Problems with Small Pages

- Small pages == Many pages
    - Pages are not located on sequential frames
        - Mechanical Hard drives incur 3 time penalties for each disk access:
        - Seek, Rotational, Transfer
        - Reading from adjacent disk location minimizes Seek and Rotational delay.
        - More pages more seek and rotational delay!

Problems with Small Pages

- Small pages == Many pages
    - Page translations are in page table
        - More pages, large page table
        - More pages, more space needed in TLB!

Small pages mean less effective TLB!

Larger Page sizes?

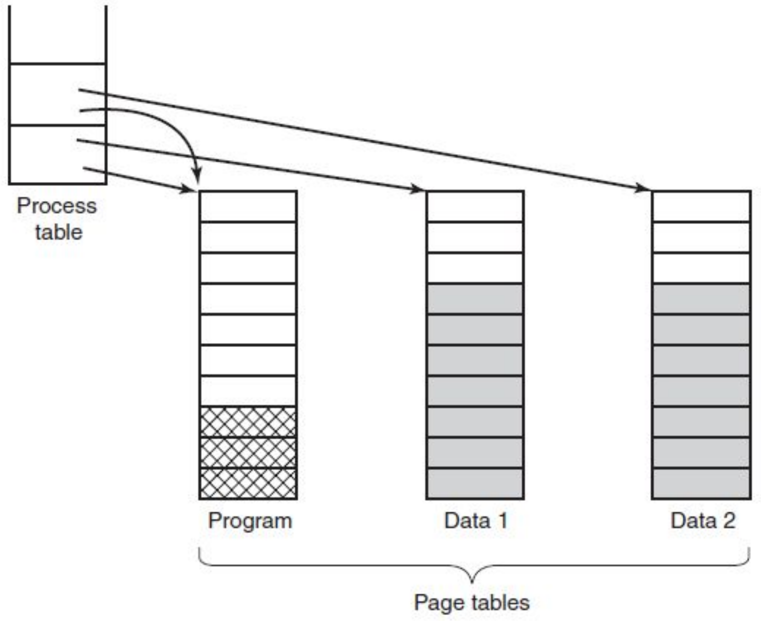
- Increase TLB Reach
- More efficient for mechanical disks
- More internal fragmentation

Good rule of thumb: More disc space, probably bigger rotational/seek delay… so use larger pages.

All pages equal?

- Often compilers lump all executable code in common pages, and all “data” in other pages.
- These pages form two “address spaces”:
  - I-space - the pages containing instructions
  - D-space - the pages containing data
  - When a compiler indicates that a page is instructions, or read-only, we can make some optimizations!

2 instances of the same process?



- Web browsers make use of this (each tab is often a separate process)
  - HTML loaded = Data
  - All the rendering and parsing = Instructions

Shared pages - implementation

- It’s easy to see how the OS facilitates this:
  - In each process’s page table, to share page “2”, simply assign it to the same frame
- Remember “shared memory” and “pipes”
  - System calls that create shared memory for processes simply create a page that is loaded to a common frame!

2 instances of the same *code*

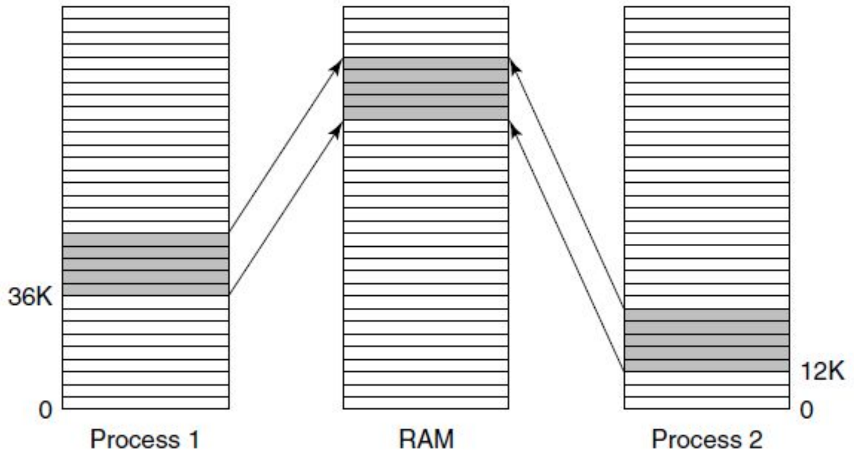
It's not just processes:

Most modern programs use *libraries*

- Some are statically compiled into the executable (STL) - which means the code must be within each process's address space
- Alternatively, many libraries are compiled as *shared libraries* **- or dynamic link libraries** (DLLs)

Sharing code pages

DLL’s are often used by different programs - and the OS can make use of this knowledge



DirectX, OpenGL, Database Drivers, etc.

Page Faults

- We’ve seen that page faults are the result of addresses being referenced on pages not loaded into frames.
    - This can happen at any time - and when using **global** allocation, a page fault in **Process A** can result in a frame being removed from **Process B**

What are the implications?

I/O - and System Calls

An operating system issues Disk reads to an I/O device by specifying a location in memory that the data should be transferred to.

- Complication:
    - What if, while being read, the destination process’s page is evicted?
    - The complexity is still present even if I/O device reads into OS buffer first - why?

Next...

We’ve seen that even with simple paging - there is a lot of complexity and tradeoffs to be considered…

Next, we look at the more complex case, where pages are not the only “grouping”: **Segmentation**