Summary of chapter 9 so far

- So far…
  - Virtual memory: extend concepts of paging a bit further.
  - Add a backing store behind main memory
    - Disk
  - Allow program to have pages that reside in memory and on disk
  - Define scheme for determining when to move pages back and forth

- Critical issues:
  - Disk is sloooooooow
  - So we want algorithms that decide to move data back and forth to the backing store as infrequently as possible
  - Page replacement algorithms
Page replacement

- Want to minimize disk<->memory transfers.

- First guess: FIFO
  - Belady’s anomaly: more frames -> more faults with FIFO
    - Bad

- Optimal (i.e., theoretically nice, but not implementable) algorithm can see future
  - Fault out page that will not be accessed for the longest period of time

- Implementable algorithms approximate this
  - First approximation: Estimate future based on past observations
  - Least Recently Used: Clock or stack can achieve this
    - But, lacking hardware assist, this is very high overhead.

- Realistic (i.e., low overhead) methods use simple bits provided by hardware
  - Referenced bit; dirty bit; “Second chance” algorithm
Frame allocation

- So we know how to manage more pages than we have frames using a virtual memory system with a backing store.

- How about frame allocation?
Allocation of Frames

- Each process needs *minimum* number of pages
  - Architecture has direct impact on this.
  - Minimum number of pages that must be available to execute all instructions.

- **Example**: IBM 370 – 6 pages to handle SS MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*

- Two major allocation schemes
  - fixed allocation
  - priority allocation
Fixed Allocation

- Equal allocation – For example, if there are 100 frames and 5 processes, give each process 20 frames.
- Proportional allocation – Allocate according to the size of process
  - $s_i$ = size of process $p_i$
  - $S = \sum s_i$
  - $m =$ total number of frames
  - $a_i =$ allocation for $p_i = \frac{s_i}{S} \times m$

$m = 64$
$s_i = 10$
$s_2 = 127$

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

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

Important! In the presence of multiprogramming, these are dynamic decisions – number of frames allocated to a process may vary as processes come and go, or grow during their lives.
Priority Allocation

- Use a proportional allocation scheme using priorities rather than size

- If process $P_i$ generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority number
Global vs. Local Allocation

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another

- **Local replacement** – each process selects from only its own set of allocated frames

- Which is better?
Global vs Local

- As usual, it depends.

- Local means that processes can be assured that aggressive, huge other processes will not cause the smaller ones to get their pages all swapped out.
  - They would pay a severe penalty performance-wise when they get a chance to run.

- Global allows big processes to get the memory they need.
  - Idle processes can’t just sit on frames that they aren’t actively using.

- Global is more commonly used in practical systems.
NUMA

- Remember NUMA?
  - Memory isn’t equal “distance” (aka, access time) from each processor.
  - We considered this before with respect to affinity.
    - Scheduler allows threads to favor processors close to their memory.

- Issue arises again with page frames.
  - NUMA -> not all frames are created equal.
  - If a process is on a processor, we might want to swap page frames into physical memory closer to the CPU where the process resides.

- This is a hard problem.
  - Only some OS’s support NUMA “well”. Solaris, IRIX, Linux.
  - These all targeted historical large NUMA SMP servers.
This machine ran IRIX:

“ASCI Blue Mountain” Big NUMA machines ca. 1999.
Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - operating system thinks that it needs to increase the degree of multiprogramming
  - another process added to the system

- **Thrashing** ≡ a process is busy swapping pages in and out

"I am probably thrashing."
Thrashing (Cont.)

![Graph showing CPU utilization vs. degree of multiprogramming]
Demand Paging and Thrashing

- Why does demand paging work?
  Locality model
  - Process migrates from one locality to another
  - Localities may overlap

- Examples of localities:
  - Functions
  - Global variables
  - Dynamically allocated and deallocated memory

- Why does thrashing occur?
  \[ \sum \text{size of locality} > \text{total memory size} \]
Locality In A Memory-Reference Pattern
Working-Set Model

- $\Delta \equiv \text{working-set window} \equiv \text{a fixed number of page references}
  \text{Example: 10,000 instruction}$
- $WSS_i (\text{working set of Process } P_i) = \text{total number of pages referenced in the most recent } \Delta \text{ (varies in time)}$
  - if $\Delta$ too small will not encompass entire locality
  - if $\Delta$ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow \text{will encompass entire program}$
- $D = \sum WSS_i \equiv \text{total demand frames}$
- if $D > m \Rightarrow \text{Thrashing}$
- Policy if $D > m$, then suspend one of the processes
Working-set model

Page reference table

\[ \ldots 2 \ 6 \ 1 \ 5 \ 7 \ 7 \ 7 \ 7 \ 5 \ 1 \ 6 \ 2 \ 3 \ 4 \ 1 \ 2 \ 3 \ 4 \ 4 \ 3 \ 4 \ 3 \ 4 \ 4 \ 4 \ 1 \ 3 \ 2 \ 3 \ 4 \ 4 \ 4 \ 3 \ 4 \ 4 \ 4 \ 4 \ldots \]

\[ WS(t_1) = \{1, 2, 5, 6, 7\} \]

\[ WS(t_2) = \{3, 4\} \]
Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: $\Delta = 10,000$
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory $= 1 \implies$ page in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units
  - Penalty: overhead for higher frequency interrupts
Overhead is very real

- Very often we see topics in here and I say:
  - “Very useful, but you pay the penalty of overhead”
  - This doesn’t mean we avoid all of these things.

- An active topic of research in OS’s
  - Understanding how this overhead interferes with applications
  - How OS’s can do their jobs with tuning and modification to reduce this interference
Page-Fault Frequency Scheme

- Establish “acceptable” page-fault rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame
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.
Memory Mapped Files

Process A virtual memory

Process B virtual memory

Physical memory

Disk file
Memory-Mapped Shared Memory in Windows

![Diagram showing memory-mapped shared memory between two processes in Windows.](image)
Allocating Kernel Memory

- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
  - Some kernel memory needs to be contiguous
    - Often when dealing with devices.
    - Devices often assume contiguous blocks of memory for things like buffers. Not page aware.
Buddy System

- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using **power-of-2 allocator**
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2
  - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
    - Continue until appropriate sized chunk available
Buddy System Allocator

physically contiguous pages

256 KB

128 KB

A_L

128 KB

A_R

64 KB

B_L

64 KB

B_R

32 KB

C_L

32 KB

C_R
Slab Allocator

- Alternate strategy
- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure
  - Each cache filled with **objects** – instantiations of the data structure
- When cache created, filled with objects marked as **free**
- When structures stored, objects marked as **used**
- If slab is full of used objects, next object allocated from empty slab
  - If no empty slabs, new slab allocated
- Benefits include no fragmentation, fast memory request satisfaction

- Linux moved from buddy system to slabs in 2.2.
- Solaris 2.4 and later used slabs. (ca. 1994)
Prepaging

- To reduce the large number of page faults that occurs at process startup
- Prepage all or some of the pages a process will need, before they are referenced
- But if prepaged pages are unused, I/O and memory was wasted
- Assume $s$ pages are prepaged and $\alpha$ of the pages is used
  - Is cost of $s \times \alpha$ save pages faults $>$ or $<$ than the cost of prepaging
    - $s \times (1 - \alpha)$ unnecessary pages?
  - $\alpha$ near zero $\Rightarrow$ prepaging loses
Other Issues – Page Size

Page size selection must take into consideration:

- fragmentation
- table size
- I/O overhead
  - Factors: seek time + latency + transfer rate
- locality
Other Issues – TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
- TLB Reach = (TLB Size) X (Page Size)
- Ideally, the working set of each process is stored in the TLB
  - Otherwise there is a high degree of page faults
- Increase the Page Size
  - This may lead to an increase in fragmentation as not all applications require a large page size
- Provide Multiple Page Sizes
  - This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation
  - Requires more software support, less hardware support
Other Issues – Program Structure

- Program structure
  - Int[128,128] data;
  - Each row is stored in one page
  - Program 1

    \[
    \text{for (j = 0; j < 128; j++)} \\
    \text{for (i = 0; i < 128; i++)} \\
    \text{data[i, j] = 0; }
    \]

    128 x 128 = 16,384 page faults

  - Program 2

    \[
    \text{for (i = 0; i < 128; i++)} \\
    \text{for (j = 0; j < 128; j++)} \\
    \text{data[i, j] = 0; }
    \]

    128 page faults

*Common issue when passing data between languages that use different nD array layouts. (Row major vs column major)*

*Dictates which indices should move fastest.*

*This can also have cache-oriented effects. Big stride per iteration may result in lots of misses.*
Other Issues – I/O interlock

- **I/O Interlock** – Pages must sometimes be locked into memory

- Consider I/O - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm
Reason Why Frames Used For I/O Must Be In Memory
Operating System Examples

- Windows XP
- Solaris
Uses demand paging with **clustering**. Clustering brings in pages surrounding the faulting page.

Processes are assigned **working set minimum** and **working set maximum**.

Working set minimum is the minimum number of pages the process is guaranteed to have in memory.

A process may be assigned as many pages up to its working set maximum.

When the amount of free memory in the system falls below a threshold, **automatic working set trimming** is performed to restore the amount of free memory.

Working set trimming removes pages from processes that have pages in excess of their working set minimum.
Solaris

- Maintains a list of free pages to assign faulting processes
- *Lotsfree* – threshold parameter (amount of free memory) to begin paging
- *Desfree* – threshold parameter to increasing paging
- *Minfree* – threshold parameter to being swapping
- Paging is performed by *pageout* process
- Pageout scans pages using modified clock algorithm
- *Scanrate* is the rate at which pages are scanned. This ranges from *slowscan* to *fastscan*
- Pageout is called more frequently depending upon the amount of free memory available