Background

• Code needs to be in memory to execute, but entire program rarely used
  • Error code, unusual routines, large data structures
• Entire program code not needed at the same time
• Consider ability to execute partially-loaded program
  • Program no longer constrained by limits of physical memory
  • Each program takes less memory while running -> more programs run at the same time
    • Increased CPU utilization and throughput with no increase in response time or turnaround time
  • Less I/O needed to load or swap programs into memory -> each user program runs faster
Background (Cont.)

- **Virtual memory** – separation of user logical memory from physical memory
  - Only a part of the program needs to be in memory for execution
  - Logical address space can be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes
Background (Cont.)

• **Virtual address space** — logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical

• Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation
Virtual Memory That is Larger Than Physical Memory
Virtual-address Space

- Usually design logical address space for stack to start at Max logical address and grow “down” while heap grows “up”
  - Maximizes address space use
  - Unused address space between the two is hole
    - No physical memory needed until heap or stack grows to a given new page
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc.
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during `fork()`, speeding process creation
Shared Library Using Virtual Memory
Swapping Using Paging

• Could bring entire process into memory at load time
• Like paging system with swapping
• No external fragmentation
Demand Paging

• Bring a page into memory only when it is needed
  • Less I/O needed, no unnecessary I/O
  • Less memory needed
  • Faster response
  • More users

• Page is needed ⇒ reference to it
  • invalid reference ⇒ abort
  • not-in-memory ⇒ bring to memory

• Lazy swapper – never swaps a page into memory unless page will be needed
  • Swapper that deals with pages is usually called a pager
Basic Concepts

• With swapping, pager guesses which pages will be used before swapping out again
• Instead, pager brings in only needed pages into memory
• How to determine that set of pages?
  • Need new MMU functionality to implement demand paging
• If pages needed are already memory resident
  • No difference from non-demand-paging
• If page needed and not memory resident
  • Need to detect and load the page into memory from storage
    • Without changing program behavior
    • Without programmer needing to change code
Valid-Invalid Bit

• With each page table entry, a valid–invalid bit is associated ($v \Rightarrow \text{in-memory} - \text{memory resident}, \ i \Rightarrow \text{not-in-memory}$)
• Initially valid–invalid bit is set to $i$ on all entries
• Example of a page table snapshot:

- During MMU address translation, if valid–invalid bit in page table entry is $i \Rightarrow$ page fault
Page Table When Some Pages Are Not in Main Memory
Page Fault

• If there is a reference to a page, first reference to that page will trap to operating system:

  \textit{page fault}

1. Operating system looks at another table to decide:
   • Invalid reference $\Rightarrow$ abort
   • Just not in memory

2. Find free frame

3. Swap page into frame via scheduled disk operation

4. Reset tables to indicate page now in memory
   Set validation bit = $v$

5. Restart the instruction that caused the page fault
Steps in Handling a Page Fault

1. Trap
2. Bring in missing page
3. Page is on backing store
4. Reset page table
5. Restart instruction
6. Load M

Diagram:
- Operating system
- Reference
- Page table
- Free frame
- Physical memory
- Load M
- Trap
- Bring in missing page
- Reset page table
- Restart instruction
- Page is on backing store
- Load M
Aspects of Demand Paging

• Extreme case – start process with no pages in memory
  • OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  • And for every other process pages on first access
  • Pure demand paging

• A given instruction could access multiple pages -> multiple page faults
  • Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory
  • Pain decreased because of locality of reference

• Hardware support needed for demand paging
  • Page table with valid / invalid bit
  • Secondary memory (swap device with swap space)
  • Instruction restart
Instruction Restart

• Must be able to restart the instruction that caused the page fault
  • Save enough state

• Consider an instruction that could access several different locations
  • block move

• auto increment/decrement location
• Restart the whole operation?
  • What if source and destination overlap?
Performance of Demand Paging

- Stages in Demand Paging (worse case)

1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk
5. Issue a read from the disk to a free frame:
   1. Wait in a queue for this device until the read request is serviced
   2. Wait for the device seek and/or latency time
   3. Begin the transfer of the page to a free frame
6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Move the page information
11. Correct the page table and other tables to show page is now in memory
12. Wait for the CPU to be allocated to this process again
13. Restore the user registers, process state, and new page table, and then resume the interrupted instruction
Performance of Demand Paging (Cont.)

• Three major activities
  • Service the interrupt – careful coding means just several hundred instructions needed
  • Read the page – lots of time
  • Restart the process – again just a small amount of time

• Page Fault Rate \( 0 \leq p \leq 1 \)
  • if \( p = 0 \) no page faults
  • if \( p = 1 \), every reference is a fault

• Effective Access Time (EAT)
  \[
  \text{EAT} = (1 - p) \times \text{memory access} \\
  + p (\text{page fault overhead} \\
  + \text{swap page out} \\
  + \text{swap page in})
  \]
Demand Paging Example

• Memory access time = 200 nanoseconds
• Average page-fault service time = 8 milliseconds
• EAT = \((1 - p) \times 200 + p \times 8,000,000\)
  = \((1 - p) \times 200 + p \times 7,999,800\)
• If one access out of 1,000 causes a page fault, then
  EAT = 8.2 microseconds.
  This is a slowdown by a factor of 40!!
• If want performance degradation < 10 percent
  • \(220 > 200 + 7,999,800 \times p\)
    \(20 > 7,999,800 \times p\)
  • \(p < .0000025\)
  • < one page fault in every 400,000 memory accesses
Demand Paging Optimizations

• Dedicate **Swap Space** in memory or Disk
  • Swap space I/O faster than file system I/O even if on the same device
    • Swap space allocated in larger chunks; less management needed than file system
  • Copy entire process image to swap space at process load time
    • Then page in and out of swap space
    • Used in older BSD Unix

• Demand page in from program binary on disk, but discard rather than paging out when freeing frame
  • Used in Solaris and current BSD
  • Still need to write to swap space
    • Pages not associated with a file (like stack and heap) – **anonymous memory**
    • Pages modified in memory but not yet written back to the file system

• Mobile systems
  • Typically, don’t support swapping
  • Instead, demand page from file system and reclaim read-only pages (such as code)
Copy-on-Write

- **Copy-on-Write** (COW) allows both parent and child processes to initially share the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated from a pool of **zero-fill-on-demand** pages
  - Pool should always have free frames for fast demand page execution
  - Don’t want to have to free a frame as well as other processing on page fault
  - Why zero-out a page before allocating it?
- `vfork()` variation on `fork()` system call has parent suspend and child using copy-on-write address space of parent
  - Designed to have child call `exec()`
  - Very efficient
Before Process 1 Modifies Page C
After Process 1 Modifies Page C
What Happens if There is no Free Frame?

• Used up by process pages
• Also, in demand from the kernel, I/O buffers, etc
• How much to allocate to each?

• Page replacement – find some page in memory, but not really in use, page it out
  • Algorithm – terminate? swap out? replace the page?
  • Performance – want an algorithm which will result in minimum number of page faults
• Same page may be brought into memory several times
Page Replacement

• Use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk

• Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory
Need For Page Replacement
Basic Page Replacement

1. Find the location of the desired page on disk

2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame,
     • use a page replacement algorithm to select a victim frame
       - Write victim frame to disk if dirty

3. Bring the desired page into the (newly) free frame; update the page and frame tables

4. Continue the process by restarting the instruction that caused the trap

Note now potentially 2 page transfers for page fault – increasing EAT
Page Replacement

1. swap out victim page
2. change to invalid
3. swap desired page in
4. reset page table for new page

valid–invalid bit

frame

page table
Page and Frame Replacement Algorithms

• **Frame-allocation algorithm** determines
  • How many frames to give each process
  • Which frames to replace

• **Page-replacement algorithm**
  • Want lowest page-fault rate on both first access and re-access

• Evaluate algorithm by running it on a string of memory references (reference string) and computing the number of page faults on that string
  • String is just page numbers, not full addresses
  • Repeated access to the same page does not cause a page fault
  • Results depend on number of frames available

• In all our examples, the **reference string** of referenced page numbers is
  \[7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1\]
Graph of Page Faults Versus The Number of Frames
First-In-First-Out (FIFO) Algorithm

- Reference string: \texttt{7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1}
- 3 frames (3 pages can be in memory at a time per process)

<table>
<thead>
<tr>
<th>Reference string</th>
<th>7 0 1 2 0 3 0 4 2 3 0 3 0 3 2 1 2 0 1 7 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page Frames</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>7 7 7 2 2 2 2 4 4 4 0 0 0 0 0 0 0 0 7 7 7</td>
</tr>
<tr>
<td>1</td>
<td>0 0 0 0 3 3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 0 0</td>
</tr>
<tr>
<td>2</td>
<td>1 1 1 1 0 0 0 3 3 3 3 3 3 2 2 2 2 1 1 1 1 0 0</td>
</tr>
<tr>
<td></td>
<td>* * * * * * * * * * * * * * * * * * * * * * *</td>
</tr>
</tbody>
</table>

15 page-faults

<table>
<thead>
<tr>
<th>Reference string</th>
<th>7 0 1 2 0 3 0 4 2 3 0 3 0 3 2 1 2 0 1 7 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page Frames</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>7 0 1 2 2 3 3 4 4 4 0 0 0 0 0 0 0 1 2 2 7 7 7</td>
</tr>
<tr>
<td>1</td>
<td>7 0 1 1 2 2 3 3 3 4 4 4 4 4 4 4 4 0 1 1 2 2 2</td>
</tr>
<tr>
<td>2</td>
<td>7 0 0 1 1 2 2 2 3 3 3 3 3 4 0 0 0 1 1 1 0 0 0 0</td>
</tr>
<tr>
<td>3</td>
<td>7 7 0 0 1 1 1 2 2 2 2 2 3 4 4 4 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

10 page-faults
First-In-First-Out (FIFO) Algorithm

<table>
<thead>
<tr>
<th>Ref string</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>1</th>
<th>2</th>
<th>5</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page Frames</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Page Fault</td>
<td>9</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ref string</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page Frames</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Page Fault</td>
<td>10</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

- Can adding more frames can cause more page faults!
  - Belady’s Anomaly
- How to track ages of pages?
  - Just use a FIFO queue
FIFO Illustrating Belady’s Anomaly

![Graph illustrating the number of page faults over the number of frames for FIFO. The graph shows a decrease in page faults as the number of frames increases, highlighting the Belady's Anomaly.]
Optimal Algorithm

• Replace page that will not be used for longest period of time
  • 9 is optimal for the example when using 3 Page Frames

• How do you know this?
  • Can’t read the future

• Used for measuring how well your algorithm performs

| Ref Str | 7 | 0 | 1 | 2 | 0 | 3 | 4 | 2 | 3 | 0 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |
PageFrame | 0 | 7 | 7 | 7 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 7 | 7 | 7 |
| 1 | 0 | 0 | 0 | 0 | 0 | 4 | 4 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 1 | 1 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 9* | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
## Optimal Algorithm

### 3 Page Frames

| Ref Str | 7 | 0 | 1 | 2 | 0 | 3 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |
| PageFrame | 0 | 7 | 7 | 7 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 7 | 7 | 7 |
|          | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 4 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|          | 2 | 1 | 1 | 1 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 1 | 1 | 1 | 1 | 1 | 1 |
|         | 9 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |

### 4 Page Frames

| Ref Str | 7 | 0 | 1 | 2 | 0 | 3 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |
| PageFrame | 0 | 7 | 7 | 7 | 7 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 7 | 7 | 7 |
|          | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|          | 2 | 1 | 1 | 1 | 1 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 1 | 1 | 1 | 1 | 1 | 1 |
|          | 3 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
|         | 8 | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * | * |
Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page

### Example

<table>
<thead>
<tr>
<th>Ref Str</th>
<th>7 0 1 2 0 3 4 2 3 0 3 2 1 2 0 1 7 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>PageFrame</td>
<td>7 0 1 2 0 3 4 2 3 0 3 2 1 2 0 1 7 0 1</td>
</tr>
<tr>
<td></td>
<td>7 0 1 2 0 3 4 2 3 0 3 2 1 2 0 1 7 0 1</td>
</tr>
<tr>
<td></td>
<td>7 0 1 2 0 3 4 2 3 0 3 2 1 2 0 1 7 0 1</td>
</tr>
<tr>
<td></td>
<td>11* * * * * * * * * * * * * * * *</td>
</tr>
</tbody>
</table>

- 11 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?
Least Recently Used (LRU) Algorithm

<table>
<thead>
<tr>
<th>Ref Str</th>
<th>7 0 1 2 0 3 4 2 3 0 3 0 3 2 1 2 0 1 7 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>PageFrame</td>
<td>7 0 1 2 0 3 4 2 3 0 3 0 3 2 1 2 0 1 7 0 1</td>
</tr>
<tr>
<td></td>
<td>7 0 1 2 0 3 4 2 3 0 3 0 3 2 1 2 0 1 7 0 1</td>
</tr>
<tr>
<td></td>
<td>7 0 1 2 0 3 4 2 3 0 3 0 3 2 1 2 0 1 7 0 1</td>
</tr>
<tr>
<td>3 Page Frames</td>
<td>11* * * * * * * * * *</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ref Str</th>
<th>7 0 1 2 0 3 4 2 3 0 3 0 3 2 1 2 0 1 7 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>PageFrame</td>
<td>7 0 1 2 0 3 4 2 3 0 3 0 3 2 1 2 0 1 7 0 1</td>
</tr>
<tr>
<td></td>
<td>7 0 1 2 0 3 4 2 3 0 3 0 3 2 1 2 0 1 7 0 1</td>
</tr>
<tr>
<td></td>
<td>7 0 1 2 0 3 4 2 3 0 3 0 3 2 1 2 0 1 7 0 1</td>
</tr>
<tr>
<td>4 Page Frames</td>
<td>8* * * * * * * *</td>
</tr>
</tbody>
</table>

2021

Copyright 2018 Silberschatz, Gavin & Gagne
LRU Algorithm (Cont.)

• Counter implementation
  • Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  • When a page needs to be changed, look at the counters to find smallest value
    • Search through table needed

• Stack implementation
  • Keep a stack of page numbers in a double link form:
  • Page referenced:
    • move it to the top
    • requires 6 pointers to be changed
  • But each update more expensive
  • No search for replacement

• LRU and OPT are cases of stack algorithms that don’t have Belady’s Anomaly
Use Of A Stack to Record Most Recent Page References

reference string

4 7 0 7 1 0 1 2 1 2 7 1 2

stack before a

stack after b
LRU Approximation Algorithms

• LRU needs special hardware and still slow

• **Reference bit**
  • With each page associate a bit, initially = 0
  • When page is referenced bit set to 1
  • Replace any with reference bit = 0 (if one exists)
    • We do not know the order, however

• **Second-chance algorithm**
  • Generally FIFO, plus hardware-provided reference bit
  • **Clock** replacement
  • If page to be replaced has
    • Reference bit = 0 -> replace it
    • reference bit = 1 then:
      • set reference bit 0, leave page in memory
      • replace next page, subject to same rules
Second-Chance (clock) Page-Replacement Algorithm

![Diagram of Second-Chance Page Replacement Algorithm]

Reference bits:
- 0
- 0
- 1
- 0
- 0
- 0
- 0
- 0
- 0
- 0
- 1
- 1
- 1

Pages:
- Circular queue of pages

Next victim:
- 1

Reference bits:
- 0
- 0
- 0
- 0
- 0
- 0
- 0
- 0
- 0
- 0
- 0
- 0

Pages:
- Circular queue of pages

Copyright 2018 Silberschatz, Gagne & Gagne
Enhanced Second-Chance Algorithm

• Improve algorithm by using reference bit and modify bit (if available) in concert
• Take ordered pair (reference, modify)
  1. (0, 0) neither recently used not modified – best page to replace
  2. (0, 1) not recently used but modified – not quite as good, must write out before replacement
  3. (1, 0) recently used but clean – probably will be used again soon
  4. (1, 1) recently used and modified – probably will be used again soon and need to write out before replacement
• When page replacement called for, use the clock scheme but use the four classes replace page in lowest non-empty class
  • Might need to search circular queue several times
Counting Algorithms

• Keep a counter of the number of references that have been made to each page
  • Not common

• **Lease Frequently Used (LFU) Algorithm:** replaces page with smallest count

• **Most Frequently Used (MFU) Algorithm:** based on the argument that the page with the smallest count was probably just brought in and has yet to be used
Page-Buffering Algorithms

• Keep a pool of free frames, always
  • Then frame available when needed, not found at fault time
  • Read page into free frame and select victim to evict and add to free pool
  • When convenient, evict victim

• Possibly, keep list of modified pages
  • When backing store otherwise idle, write pages there and set to non-dirty

• Possibly, keep free frame contents intact and note what is in them
  • If referenced again before reused, no need to load contents again from disk
  • Generally useful to reduce penalty if wrong victim frame selected
Applications and Page Replacement

• All of these algorithms have OS guessing about future page access
• Some applications have better knowledge – e.g., databases
• Memory intensive applications can cause double buffering
  • OS keeps copy of page in memory as I/O buffer
  • Application keeps page in memory for its own work
• Operating system can give direct access to the disk, getting out of the way of the applications
  • Raw disk mode
• Bypasses buffering, locking, etc.
Allocation of Frames

• Each process needs _minimum_ number of frames

• 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

• _Maximum_ of course is total frames in the system

• Two major allocation schemes
  • fixed allocation
  • priority allocation

• Many variations
Fixed Allocation

• Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
  • Keep some as free frame buffer pool

• Proportional allocation – Allocate according to the size of process
  • Dynamic as degree of multiprogramming, process sizes change
    \[
    \begin{align*}
    s_i &= \text{size of process } p_i \\
    S &= \sum s_i \\
    m &= \text{total number of frames} \\
    a_i &= \text{allocation for } p_i = \frac{s_i}{S} \times m
    \end{align*}
    \]
    \[
    \begin{align*}
    m &= 64 \\
    s_1 &= 10 \\
    s_2 &= 127 \\
    a_1 &= \frac{10}{137} \times 62 = 4 \\
    a_2 &= \frac{127}{137} \times 62 = 57
    \end{align*}
    \]
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
  • But then process execution time can vary greatly
  • But greater throughput so more common

• **Local replacement** – each process selects from only its own set of allocated frames
  • More consistent per-process performance
  • But possibly underutilized memory
Non-Uniform Memory Access

• So far, all memory accessed equally (Same access time)
• Many systems are **NUMA** – speed of access to memory varies
  • Consider system boards containing CPUs and memory, interconnected over a system bus
• Optimal performance comes from allocating memory “close to” the CPU on which the thread is scheduled
  • And modifying the scheduler to schedule the thread on the same system board when possible
• Solved by Solaris by creating **lgroups**
  • Structure to track CPU / Memory low latency groups
  • Used my schedule and pager
  • When possible, schedule all threads of a process and allocate all memory for that process within the lgroup
Virtual Memory

• Paging
  • Demand paging
  • Page Replacement Algorithms
    • FIFO, Optimal, LRU
    • Stack Algorithms
  • Implementations
    • Approximations
  • Strategies
    • Global vs. local
Thrashing

• If a process does not have “enough” pages, the page-fault rate is very high
  • Page fault to get page
  • Replace existing frame
  • But quickly need replaced frame back
  • This leads to:
    • Low CPU utilization
    • Operating system thinking 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
Thrashing (Cont.)
Demand Paging and Thrashing

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

• Why does thrashing occur?
  ∑ size of locality > total memory size
  • Limit effects by using local or priority page replacement
Locality In A Memory-Reference Pattern

Working set at time a
\{18, 19, 20, 21, 22, 23, 24, 29, 30, 33\}

Working set at time b
\{18, 19, 20, 24, 25, 26, 27, 28, 29, 31, 32, 33\}
Working-Set Model

- $\Delta \equiv \text{working-set window} \equiv \text{a fixed number of page references}$
  - Example: 10,000 instructions

- $WSS_i$ (working set of Process $P_i$) =
  - total number of pages referenced in the most recent $\Delta$ (varies in time)
    - if $\Delta$ too small will not encompass entire locality
    - if $\Delta$ too large will encompass several localities
    - if $\Delta = \infty \implies$ will encompass entire program

- $D = \Sigma WSS_i \equiv \text{total demand frames}$
  - Approximation of locality
    - if $D > m \implies$ Thrashing
    - Policy if $D > m$, then suspend or swap out one of the processes
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 reference bits to memory
    • Set all reference bits to 0
    • If one of the bits for a page (in memory or reference bit) is 1
      $\Rightarrow$ page in working set
• Why is this not completely accurate?
• Improvement = 10 bits and interrupt every 1000 time units
Page-Fault Frequency

• More direct approach than WSS
• Establish “acceptable” page-fault frequency (PFF) rate and use local replacement policy
  • If actual rate too low, process loses frame
  • If actual rate too high, process gains frame
Working Sets and Page Fault Rates

1. Direct relationship between working set of a process and its page-fault rate
2. Working set changes over time
3. Peaks and valleys over time
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 and speeds file access by driving file I/O through memory rather than read() and write() system calls
• Also allows several processes to map the same file allowing the pages in memory to be shared
• But when does written data make it to disk?
  • Periodically and / or at file close() time
  • For example, when the pager scans for dirty pages
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
    • I.e., for device I/O
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 appropriately sized chunk available

• For example, assume 256KB chunk available, kernel requests 21KB
  • Split into $A_L$ and $A_R$ of 128KB each
    • One further divided into $B_L$ and $B_R$ of 64KB
      • One further into $C_L$ and $C_R$ of 32KB each – one used to satisfy request

• Advantage – quickly **coalesce** unused chunks into larger chunk

• Disadvantage - fragmentation
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
  - (this is not the hardware memory cache of the cpu)
- Single cache for each unique kernel data structure
  - Each cache filled with **objects** – instantiations of the data structure
  - For example – Cache representing semaphores stores instances of semaphore objects
- 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
Slab Allocation
Slab Allocator in Linux

- For example, process descriptor is of type `struct task_struct`
- Approx 1.7KB of memory
- New task -> allocate new struct from cache
  - Will use existing free `struct task_struct`
- Slab can be in three possible states
  1. Full – all used
  2. Empty – all free
  3. Partial – mix of free and used
- Upon request, slab allocator
  1. Uses free struct in partial slab
  2. If none, takes one from empty slab
  3. If no empty slab, create new empty
Slab Allocator in Linux (Cont.)

• Slab started in Solaris, now wide-spread for both kernel mode and user memory in various OSes

• Linux 2.2 had SLAB, now has both SLOB and SLUB allocators
  • SLOB for systems with limited memory
    • Simple List of Blocks – maintains 3 list objects for small, medium, large objects
  • SLUB is performance-optimized SLAB removes per-CPU queues, metadata stored in page structure
Other Considerations -- Prepaging

• 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 a fraction of the pages is used
    • Is cost of saved pages faults > or < than the cost of prepaging unnecessary pages?
Other Issues – Page Size

• Sometimes OS designers have a choice
  • Especially if running on custom-built CPU

• Page size selection must take into consideration:
  • Fragmentation
  • Page table size
  • Resolution
  • I/O overhead
  • Number of page faults
  • Locality
  • TLB size and effectiveness

• Always power of 2, usually in the range $2^{12}$ (4,096 bytes) to $2^{22}$ (4,194,304 bytes)

• On average, growing over time
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
Other Issues – Program Structure

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

    for (j = 0; j < 128; j++)
    for (i = 0; i < 128; i++)
        data[i,j] = 0;

    128 x 128 = 16,384 page faults

• Program 2

    for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i,j] = 0;

    128 page faults
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

• **Pinning** of pages to lock into memory
Operating System Examples

• Linux

• Windows

• Solaris
4 BSD UNIX

• Global replacement policy
  • Using second chance algorithm
• Implemented by *pageout daemon*
  • Also uses a *swapper* process
  • Both execute in kernel mode with their own process structure and kernel stack
• There is a **core map table (cmap)** with one entry for each page frame
• Executing process has its page table in the memory
4 BSD UNIX- Page Replacement Algorithm

• Clock Algorithm implemented by *pageout daemon*
  • A software clock hand sweeps all entries in cmap
    • If the frame is unused, advance the clock hand to the next entry.
    • If I/O is active on the page, leave it as it is.
    • If the reference bit is set, reset it.
    • If the reference bit is reset (because the page have not been referenced since last time the bit was reset), release the page by entering it into free list.
      • If the page was modified, it must first be written to the paging disk.
4 BSD UNIX- Clock Algorithm with two Clock Hands

• If a large main memory is used, the clock algorithm with one hand does not work very well because the time to scan all page frames is too large.

• On some systems an algorithm with two clock hands is used.

• The first clock hand is used to reset the reference bit, and the second clock hand releases pages with the referenced bit still reset. (they have not been referenced in the time span between the scan of the first and second clock hands)
Page Replacement Algorithm for two Clock Hands

- The goal for the pageout daemon is to assure that there are a sufficient number of free frames available at all times in a free frame list.
- The pageout daemon is awakened every 250 ms by a timeout to check that there is at least \texttt{lotsfree} (system parameter) free page frames. If this is the case the pageout daemon goes back to sleep.
- If the number of free frames is below \texttt{lotsfree}, the clock hand sweeps a number of steps. How many steps depends on the number of pages needed to reach \texttt{lotsfree} free frames.
- There is a maximum number of steps the pageout daemon is allowed to sweep at one occasion. This is adjusted so that the pageout daemon should not use more than 10% of the CPU time.
4 BSD UNIX - Swapping

If it is discovered that the paging system is overloaded, the swapper process is started to swap out some processes entirely (including page table and u-block). The reason for using swapping is to try to avoid that the system enters a thrashing condition.

The swapper process is started only if the following conditions are fulfilled:

• 1. Load average is high (many processes in the ready queue).
• 2. The number of free page frames is below a low value minfree.
• 3. The average number of free frames is less than desfree.
  • (lotsfree > desfree > minfree).
4 BSD UNIX - Swapping

Swapout:

• If some process has been idle more than 20 seconds, swap it out.
• Else select the one among the four biggest processes that has been in main memory the longest time.
• The algorithm is repeated until a sufficient number of frames is available or no more candidate processes for swapping can be found.

Swapin

• With a few seconds interval, the swapper process checks if some swapped out process can be swapped in again. Only the page table and the u-block is brought in by the swapper process. The pages are not fetched until they generate a page fault.
Linux

- Kernel memory – slab allocation
- Rest – uses demand paging
  - Global page replacement policy
    - LRU approximation clock algorithm
  - Maintains two lists
    - Active List – pages considered to be in use
    - Inactive List – pages that have not recently been referenced and are eligible to be reclaimed
Linux

- Accessed bit
  - Set when first allocated
  - Added to the rear of the active_list
  - When referenced access bit set and moved to the rear of the Active_list
- Periodically access bits for pages in the active_list are reset
- Pages from the front of the Active_list may move to the rear of the Inactive_list
- If a page on inactive_list is referenced it is moved to active_list
- Free page List
  - Paging daemon – *kswapd*
  - Awakens and if the size of Free page list below a threshold
    - Reclaims the pages from the front of Inactive_list
Windows

• 32 bit architecture
  • Virtual address space 2 GB – can be extended to 3 GB
  • Physical memory – up to 4 GB

• 64 bit architecture
  • 128 TB Virtual address space
  • Up to 24 TB physical memory (server version supports 128 TB)
Windows

• 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 as possible 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

- Assigns page from the list of free pages
- 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
- **Priority paging** gives priority to process code pages
Solaris 2 Page Scanner

![Diagram showing the relationship between scan rate and amount of free memory. At the top, fastscan at 8192. At the bottom, slowscan at 100. On the x-axis, minfree, desfree, and lotsfree represent different levels of free memory.]