CS370 Operating Systems
Colorado State University
Yashwant K Malaiya
Fall 2021 Lecture 19

Virtual Memory

Slides based on
- Text by Silberschatz, Galvin, Gagne
- Various sources
Questions from last time

• TLB:
  – serves as a cache for Page Table
  – Small subset of the info in Page Table, but high hit rate
  – Can be multilevel, *may be separate for data/instruction*
  – How to find TLB hit rate? (Answer: simulation)

• “64-bit” chips:
  – Within the CPU, data/addresses are mostly 64 bit.
  – Externally addresses may be 48 bits. Things are not straightforward these days.

• Comment on the terminology in Operating Systems
  – Terms coined by developers of various schemes at different times
  – Terms like “TLB”, “Hadoop” etc.
Questions from last time

Page table: Separate page table for each process

- Index: page number (used as an address); entry: frame number.
- Page table needs to occupy contiguous memory locations. Problem when p has too many bits (Solution: use multi-level page tables.)

\[ p: \text{page number} \]
\[ f: \text{frame number} \]

Page size is the same as frame size

Where are the frames?
How big is a frame?
How many frames?

Frames are in memory.
A frame is \(2^{12}\) bytes.
Up to \(2^{20}\) frames.
Two-Level Page-Table Scheme

Outer page table: $2^{p_1} = 2^{12}$ entries
- entry points to beginning of a page in the page table

Page Table: with $2^{12}$ pages, each with $2^{p_2} = 2^{10}$ entries
- Entry points to a frame in physical memory

Physical memory: Many frames. $d$ is the offset within the frame of size $2^d = 2^{10}$
Demand paging: Basic Concepts

- Demand paging: pager brings in only those pages into memory what are needed
- 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 $ in-memory – memory resident, $i \Rightarrow $ not-in-memory)
- Initially valid–invalid bit is set to $i$ on all entries
- Example of a page table snapshot:

  ![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 0 in Frame 4 (and disk)
Page 1 in Disk
Page Fault

- If there is a reference to a page, first reference to that page will trap to operating system: Page fault

**Page fault**

1. Operating system looks at a table to decide:
   - Invalid reference $\Rightarrow$ abort
   - Just not in memory, but in *backing storage*, $\Rightarrow$2
2. Find free frame
3. Get 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

Page fault: context switch because disk access is needed
Technical Perspective: Multiprogramming

Solving a problem gives rise to a new class of problem:

- Contiguous allocation. **Problem**: external fragmentation
- Non-contiguous, but entire process in memory: **Problem**: Memory occupied by stuff needed only occasionally. Low degree of Multiprogramming.
- Demand Paging: **Problem**: page faults
- How to minimize page faults?
Steps in Handling a Page Fault

1. Trap
2. Bring in missing page
3. Page is on backing store
4. Reset page table
5. Free frame
6. Restart instruction
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. **Correct the page table and other tables to show page is now in memory**
11. Wait for the CPU to be allocated to this process again
12. 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 – relatively long 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)
  $EAT = (1 - p) \times memory\ access\ time$
  $\quad + p (page\ fault\ overhead$
  $\quad \quad + swap\ page\ out\ +\ swap\ page\ in )$

Hopefully $p << 1$

Page swap time = seek time + latency time
Demand Paging Simple Numerical Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = \((1 - p) \times 200 \text{ ns} + p \times 8,000,000 \text{ nanosec.}\)
  = \(200 + p \times 7,999,800 \text{ ns}\)

- 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, \(p = ?\)
  \[220 > 200 + 7,999,800 \times p\]
  \[20 > 7,999,800 \times p\]
  \[p < 0.0000025\]
  \(< \text{one page fault in every 400,000 memory accesses}\)

We make some simplifying assumptions here.
Issues: Allocation of physical memory to I/O and programs

- Memory used for holding **program** pages
- **I/O buffers** also consume a big chunk of memory
- Solutions:
  - Fixed percentage set aside for I/O buffers or
  - Processes and the I/O subsystem compete
Demand paging and the limits of logical memory

• Without demand paging
  – All pages of process **must be** in physical memory
  – Logical memory **limited** to size of physical memory

• With demand paging
  – All pages of process **need not be** in physical memory
  – Size of logical address space is **no longer constrained** by physical memory

• Example
  – 40 pages of physical memory
  – 6 processes each of which is 10 pages in size
    • But each process only needs 5 pages **as of now**
  – Run 6 processes with 10 pages to spare

**Higher degree of multiprogramming**
Coping with over-allocation of memory

Example

- Physical memory = 40 pages
- 6 processes each of which is of size 10 pages
  - But are using 5 pages each as of now
- What happens if each process needs all 10 pages?
  - 60 physical frames needed

- **Option: Terminate** a user process
  - But paging should be transparent to the user

- **Option: Swap out** a process
  - Reduces the degree of multiprogramming

- **Option: Page replacement**: selected pages. Policy?
• **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 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? *(security)*
Copy-on-write

Before Process 1 Modifies Page C

After Process 1 Modifies Page C
What Happens if there is no Free Frame?

• Could be all used up by process pages or 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

Continued to Page replacement etc...
Page Replacement

- Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory
- Use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk
Basic Page Replacement

1. Find the location of the desired page on disk
2. Find a free frame:
   I. If there is a free frame, use it
   II. If there is no free frame, use a page replacement algorithm to select a victim frame
   III. 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

Page table after swap

1. Swap out victim page
2. Change to invalid
3. Swap desired page in
4. Reset page table for new page
More algorithms ...

Jim unwittingly wanders into a rough section of the Computer Science department.
• **Page-replacement algorithm**
  – Which frames to replace
  – Want lowest page-fault rate

• **Evaluate algorithm** by running it on a particular 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, we use 3 frames, and the **reference string** of referenced page numbers is
  
  7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1
Graph of Page Faults Versus The Number of Frames

What we would generally expect
Page Replacement Algorithms

Algorithms

• FIFO
• “Optimal”
• The Least Recently Used (LRU)
  – Exact Implementations
    • Time of use field, Stack
  – Approximate implementations
    • Reference bit
    • Reference bit with shift register
    • Second chance: clock
    • Enhanced second chance: dirty or not?

• Other
FIFO page replacement algorithm: Out with the old; in with the new

- When a page must be replaced
  - Replace the oldest one

- OS maintains list of all pages currently in memory
  - Page at head of the list: Oldest one
  - Page at the tail: Recent arrival

- During a page fault
  - Page at the head is removed
  - New page added to the tail
First-In-First-Out (FIFO) Algorithm

- Reference string:
  \[7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1\]
- 3 frames (3 pages can be in memory at a time per process)
- 15 page faults (out of 20 accesses)
- Sometimes a page is needed soon after replacement  7,0,1,2,0,3 (0 out),0, ..
Belady’s Anomaly

• Consider Page reference string 1,2,3,4,1,2,5,1,2,3,4,5
  – 3 frames, 9 faults, 4 frames 10 faults! Try yourself.
  – Sometimes adding more frames can cause more page faults!

  • Belady’s Anomaly

Lazlo Belady was here at CSU. Guest in my CS530!

Budapest, 1928
“Optimal” Algorithm

• Replace page that will not be used for longest period of time

- 4th access: replace 7 because we will not use if got the longest time...
- 9 page replacements is optimal for the example

• But how do we know the future pages needed?
  - Can’t read the future in reality.

• Used for *measuring* how well an algorithm performs.
Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time (4\textsuperscript{th} access – page 7 is least recently used ...)
- Associate time of last use with each page

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

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

- 12 faults – better than FIFO (15) but worse than OPT (9)
- Generally good algorithm and frequently used
- But how to implement it by tracking the page usage?

LRU and OPT are cases of stack algorithms that don’t have Belady’s Anomaly
Least Recently Used (LRU) Algorithm

LRU page number is marked (*). Unmarked if that page is accessed.

LRU applied to cache memory.
Least Recently Used (LRU) Algorithm

* Use past knowledge rather than future

- 12 faults – better than FIFO (15) but worse than OPT (9)

- Tracking the page usage. One approach: mark least recently used page each time.

|   | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |
| 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |
| 7 | 7* | 7* | 2 | 2 | 2* | 4 | 4 | 4* | 0 | 0 | 0* | 1 |   |   |   |   |   |   |   |
| 0 | 0 | 0* | 0 | 0 | 0 | 0 | 0* | 3 | 3 | 3 | 3 |   |   |   |   |   |   |   |
|   | 1 | 1 | 1* | 3 | 3 | 3* | 2 | 2 | 2 | 2* | 2 | 2 |   |   |   |   |   |   |   |

- Other approach: use stack for tracking (soon)
CS370 Operating Systems

Colorado State University

Yashwant K Malaiya

Back from ICQ
Possible tracking implementations

• 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
  – Each update expensive
  – No search for replacement needed (bottom is least recently used)
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

Most recently used ->

2
1
0
7
4

stack before
a

7
2
1
0
4

stack after
b

Least recently used ->

This shows tracking stack, not actual frames.

Too slow if done in software
Use Of A Stack to Record Most Recent Page References

Examine this at home.

<table>
<thead>
<tr>
<th>4</th>
<th>7</th>
<th>0</th>
<th>7</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>1</th>
<th>2</th>
<th>7</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>7</td>
<td>0</td>
<td>7</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>7</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>0</td>
<td>7</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>7</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Most recently used ->

<table>
<thead>
<tr>
<th>Least recently used -&gt;</th>
</tr>
</thead>
</table>

Detailed version of previous slide.
This shows tracking stack, not actual frames.
Use Of A Stack to Record Most Recent Page References

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

Earlier problem (upper) revisited.
This shows tracking stack, not actual frames.

<table>
<thead>
<tr>
<th>page frames</th>
<th>7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MRU-/&gt;</td>
<td>7 0 1 2 0 3 0 4 2 3 0 3 0 4 2 3 0 4 2 3 0</td>
</tr>
<tr>
<td>LRU-/&gt;</td>
<td>7 0 1 2 2 3 0 4 2 2 3 0 4 2 3 0 4 2 3 0</td>
</tr>
</tbody>
</table>
LRU Approximation Algorithms

• LRU needs special hardware and still slow

• **Reference** 1 bit per frame to track history
  - With each page associate a bit, initially = 0
  - When the page is referenced, bit set to 1
  - Replace any page with reference bit = 0 (if one exists)
    - 0 implies not used since initialization
    - We do not know the order, however.

• Advanced schemes using more bits: preserve more information about the order
Ref bit + history shift register

LRU approximation 9 bits per frame to track history

Ref bit: 1 indicates used, Shift register records history. Examples:

<table>
<thead>
<tr>
<th>Ref Bit</th>
<th>Shift Register</th>
<th>Shift Register after OS timer interrupt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0000 0000</td>
<td>1000 0000</td>
</tr>
<tr>
<td>1</td>
<td>1001 0001</td>
<td>1100 1000</td>
</tr>
<tr>
<td>0</td>
<td>0110 0011</td>
<td>0011 0001</td>
</tr>
</tbody>
</table>

- Interpret 8-bit bytes as **unsigned integers**
- Page with the lowest number is the LRU page: replace.

Examples:
- 00000000 : Not used in last 8 periods
- 01100101 : Used 4 times in the last 8 periods
- 11000100 used more recently than 01110111
Second-chance algorithm

- Generally FIFO, plus hardware-provided reference bit
- Avoid throwing out a heavily used page
- “Clock” replacement (using circular queue): hand as a pointer
- Consider next page
  - Reference bit = 0 -> replace it
  - reference bit = 1 then: give it another chance
    - set reference bit 0, leave page in memory
    - consider next page, subject to same rules
Second-Chance (clock) Page-Replacement Algorithm

- **Clock** replacement: hand as a pointer
- Consider next page
  - Reference bit = 0 -> replace it
  - Reference bit = 1 then:
    - set reference bit 0, leave page in memory
    - consider next page, subject to same rules

Example:
(a) Change to 0, give it another chance
(b) Already 0. Replace page
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

• **Least 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
Clever Techniques for enhancing Perf

• Keep a buffer (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
• Keep list of modified pages
  – When backing store is otherwise idle, write pages there and set to non-dirty (being proactive!)
• Keep free frames’ previous 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
Buffering and applications

• Some applications (like databases) often understand their memory/disk usage better than the OS
  – Provide their own buffering schemes
  – If both the OS and the application were to buffer
    • Twice the I/O is being utilized for a given I/O
  – OS may provide “raw access” disk to special programs without file system services.
Allocation of Frames
How to allocate frames to processes?

- Each process needs *minimum* number of frames
  Depending on specific needs of the process
- *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 (need based)
  - Dynamic as degree of multiprogramming, process sizes change

\[
s_j = \text{size of process } p_j \\
S = \sum s_j \\
m = \text{total number of frames} \\
a_j = \text{allocation for } p_j = \frac{s_j}{S} \times m
\]

Example: Processes P1, P2

\[
m = 62 \\
s_1 = 10 \\
s_2 = 127 \\
a_1 = \frac{10}{137} \times 62 \approx 4 \\
a_2 = \frac{127}{137} \times 62 \approx 57
\]
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 or
  – 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
Problem: 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, leading to
    • Operating system thinking that it needs to increase the degree of multiprogramming leading to
    • Another process added to the system

• **Thrashing** ≡ a process is busy swapping pages in and out
Thrashing (Cont.)

![Graph showing CPU utilization vs degree of multiprogramming with a shaded area indicating thrashing.](image-url)
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 in a process?
  
  size of locality > total memory size allocated

  - Limit effects by using local or priority page replacement
Locality In A Memory-Reference Pattern
Working-Set Model

- \( \Delta \equiv \text{working-set window} \equiv \) a fixed number of page references

Example: \( \Delta = 10 \) page references

- \( WSS_i \) (working set of Process \( P_i \)) =
  - total number of pages referenced in the most recent \( \Delta \) (varies in time)
    - if \( \Delta \) too small, working set will not encompass entire locality
    - if \( \Delta \) too large, working set will encompass several localities
    - \( ws \) is an approximation of locality

- \( D = \sum WSS_i \equiv \text{total demand for frames} \) for all processes
  - if \( D > m \) \( \Rightarrow \) Thrashing
  - Policy if \( D > m \), then suspend or swap out one of the processes
Page-Fault Frequency Approach

• More direct approach than WSS
• Establish “acceptable” page-fault frequency (PFF) rate for a process 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

- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
- Peaks and valleys over time

Peaks occur at locality changes: 3 working sets
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
• File is then in memory instead of disk
• 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
Memory Mapped Files

Disk File uses 6 blocks
Page tables used for mapping
Allocating Kernel Memory

• Treated differently from user memory
• Often allocated from a free-memory pool
  – Kernel requests memory for structures of varying sizes
    • Process descriptors, semaphores, file objects etc.
    • Often much smaller than page size
  – Some kernel memory needs to be contiguous
    • e.g. for device I/O
  – approaches (skipped)
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 fraction $\alpha$ of the pages is used
    • Is cost of $s \times \alpha$ saved pages faults $>$ or $<$ than the cost of prepaging $s \times (1-\alpha)$ unnecessary pages?
    • $\alpha$ near zero $\Rightarrow$ greater prepaging loses
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
  – 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
Page size 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
Other Issues – Program Structure

- **Program structure**
  - \( \text{int}[128,128] \) data; \( i \): row, \( j \): column
  - 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  inner loop = 1 row = 1 page
  ```
  for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
      data[i,j] = 0;
  ```
  128 page faults