• Last class:
  – Paging

• Today:
  – Virtual Memory
Virtual Memory

- What if programs require more memory than available physical memory?
  - Use overlays
    - Difficult to program though!
  - Virtual Memory.
    - Supports programs that are larger than available physical memory.
    - Allows several programs to reside in physical memory (or at-least the relevant portions of them).
    - Allows non-contiguous allocation without making programming difficult.
Example

Physical Memory
(16 KB)

Virtual Address Space-1 (1 MB)

Virtual Address Space-2 (1 MB)

Virtual Address Space-n (1 MB)
Page Faults

• If a Page-table mapping indicates an absence of the page in physical memory, hardware raises a “Page-Fault”.

• OS traps this fault and the interrupt handler services the fault by initiating a disk-read request.

• Once page is brought in from disk to main memory, page-table entry is updated and the process which faulted is restarted.
  – May involve replacing another page and invalidating the corresponding page-table entry.
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:
  – page fault
• Operating system looks at another table to decide:
  – Invalid reference -- abort
  – Just not in memory
• Get empty frame
• Swap page into frame
• Reset tables
• Set validation bit = v
• Restart the instruction that caused the page fault
Steps in Handling a Page Fault

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

Load M
Performance of Demand Paging

• Page Fault Rate
  – $0 \leq p \leq 1.0$
  – if $p = 0$ no page faults
  – if $p = 1$, every reference is a fault

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

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds

\[
EAT = (1 - p) \times 200 + p \times 8 \text{ milliseconds}
\]
\[
= (1 - p \times 200 + p \times 8,000,000
\]
\[
= 200 + p \times 7,999,800
\]

- If one access out of 1,000 causes a page fault, then
  \[EAT = 8.2 \text{ microseconds.}\]
  This is a slowdown by a factor of 40!!
Putting it all together!

int A[2K];
int B[2k];

main() {
    int i, j, p;
    p = malloc(16K);
}

4K page size
After executing malloc

Malloc/free first manipulate this space (using buddy, ...)

If out of Space, call OS To allocate more PT entries (using sbrk())

Note: you are not allocating physical memory using malloc()
Page Replacement

• When bringing in a page, something has to be evicted.

• What should we evict? – page replacement algorithm.
Optimal Page Replacement Algorithm

• Why optimal?
  – No other algorithm can have # of page faults lower than this, for a given page reference stream.

• Algorithm:
  – At any point, amongst the given pages in memory, evict the one whose next reference from now is the furthest in the future.
An example of OPT

Reference String

\[ \ldots, 5, 3, 3, 5, 2, 4, 4, 3, 2, \ldots \]

At this point, what do we replace?

Current Physical Memory

\[
\begin{array}{c}
5 \\
3 \\
4 \\
2
\end{array}
\]

Evict
Problem with OPT

• Not implementable!

• Requires us to know the future.

• But it has the best page fault behavior

• How do we approach OPT?
1. First-in First-out

- Maintain a linked list of pages in the order they were brought into PM.
- On a page fault, evict the one at the head.
- Put the newly brought in page (from disk) at tail of this list.
- Problems:
  - Reference String: 1,2,3,4,1,1,5,1,1,…
  - Page fault at (5) would replace (1)!
  - Need to know what is in recent use!
2. Not Recently Used

- Referenced bit set on each Read/write by h/w
- Modified set on each write by h/w
- On startup set both R and M bits to 0.
- Periodically (using clock interrupts) the R bit is cleared.

- On a page fault, examine the state of a page
  - Class 0: \( R = M = 0 \)
  - Class 1: \( R = 0, M = 1 \)
  - Class 2: \( R = 1, M = 0 \)
  - Class 3: \( R = 1, M = 1 \)

- NRU replaces a page chosen at random from the lowest numbered nonempty class.
3. Second Chance Replacement or Clock Algorithm

• Same as FIFO, except you skip over the pages whose reference bit is set, resetting this bit, and moving those pages to end of list.

• Implementation:
4. Least Recently Used

- Order the list of physical memory pages in decreasing order of recency of usage.
- Replace the page at the tail.
- Problem:
  - This list will need to be updated on each memory reference.
  - Asking the h/w to do this is ridiculous!
- Solution: Approximate LRU
5. Approximate LRU using counters

• Keep a counter for each Phys page.
• Initially set to 0.
• At the end of each time interval (interval to be determined), shift the bits right by one position.
• Copy the reference bit to the MSB of counter and reset reference.
• For a page replacement, pick the one with the lowest counter value.
• It is an approximation of LRU because:
  – we do not differentiate between references that occurred in the same tick.
  – the history is limited by the size of the counter.
Summary of page replacement algorithms

• OPT, FIFO, NRU, second-chance/clock, LRU, approximate LRU

• In practice, OSes use second chance/clock or some variations of it.
Belady’s Anomaly

- Normally you expect number of page faults to decrease as you increase physical memory size.

- However, this may not be the case in certain replacement algorithms.
Example of Belady’s Anomaly

• FIFO replacement Algorithm
• Reference string:
  0 1 2 3 0 1 4 0 1 2 3 4
• 3 physical frames
  F F F F F F F - - F F -
  # of faults = 9
• 4 physical frames
  F F F F - - F F F F F F
  # of faults = 10
• Algorithms which do NOT suffer from Belady’s anomaly are called stack algorithms
• E.g. OPT, LRU.
Modeling Paging

• Paging behavior characterized by
  – Reference string
  – Physical memory size
  – Replacement algorithm
• Visualize it as a stack (say $M$), where a page that is “referenced” is brought to the top of the stack from wherever it is.

e.g. A, B, C and D are virtual pages

When C is referenced ...
• Whatever is in recent use is on the top of M, and the ones that are not in recent use are at the bottom.
• In fact, the top P entries of M represent the pages in physical memory, where P is the # of physical frames.
• **Distance String:**
  
  - For each element of reference string, this represents the distance of that element from the top of stack in M.
An example of how $M$ changes with 5 virtual pages

<table>
<thead>
<tr>
<th>Reference String</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>D</td>
</tr>
<tr>
<td>∞</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Distance String</th>
</tr>
</thead>
<tbody>
<tr>
<td>∞</td>
</tr>
</tbody>
</table>
Define vector $C$

- $C[i]$ represents the number of times “$i$” appears in the distance string.
### Reference String

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>E</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>A</td>
<td>B</td>
<td>E</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
</tr>
</tbody>
</table>

### Distance String

3 3 ∞ 2 2 4 4 4

### C vector:

- $C[0]=0$, $C[1]=0$, $C[2]=2$,
- $C[\infty]=5$
Define Vector F

- \( F[j] \) is the number of page faults that will occur for the given reference string with “j” physical frames.

• It is now straightforward to prove LRU does not suffer from Belady’s anomaly.
  – The M vector tracks what is in physical memory in the top P slots for LRU.
  – Note that vector C[i] is independent of physical memory size.
  – When you go from physical memory with j frames to (j +x) frames, note that the number of C vector terms in the RHS of equation for F decreases => Page faults can only decrease if at all!
Paging Issues

• Keep the essentials of what you currently need (working set) in physical memory.
• When something you need is not in memory, bring it in from disk:
  – On demand (demand-paging)
  – Ahead of need (pre-paging)
• Programs need to exhibit good locality to avoid “thrashing” of pages in memory.
• This usually requires good programming skills!
Fragmentation in paging

• Note that there is only internal fragmentation, and that too only in the last allocated page.

• Smaller the page, smaller the internal fragmentation.

• However, this reduces spatial locality.
Page size trade-offs

• Average process size = $s$ bytes
• Page size = $p$ bytes
• Page Table entry = $e$ bytes

• Overhead = $s.e/p + p/2$
• To minimize, $p = \sqrt{2.s.e}$
Summary

• Page Replacement
  – Virtual memory
  – Page faults
  – Optimal page replacement not achievable
  – Variety of algorithms
  – Anomalies
• Next time: Virtual memory issues