Today: Demand Paged Virtual Memory

- Segmented Paging: combine the best features of paging and segmentation
  - Each program consists of a number of segments
  - User view of programs
- The illusion of an infinite virtual memory enables much slower, cache or set of registers
- main memory
  - L. treat disk (or other backing store) as a much larger, but much slower OS illusions:
  - assumed it was all in memory
  - Up to now, the virtual address space of a process fit in memory, and we

Segmented Paging: combine the best features of paging and segmentation
**Demand Paged Virtual Memory: Example**

<table>
<thead>
<tr>
<th>Disk</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page Table</td>
<td>Page Table</td>
</tr>
<tr>
<td></td>
<td>Virtual Memory</td>
</tr>
</tbody>
</table>

**Key Ideas:**

- **Locality** — the working set size of a process must fit in memory, and must stay there.

  - For efficiency reasons, memory access time must be low. For this reason, memory access must reference pages that are in memory. The vast majority of the time, the effective memory access time will approach that of the disk.

  - Else the effective memory access time will approach that of the disk.

**Once a page is brought from disk into memory, the OS updates the page table and the valid bit.**

**Using a valid bit:**

- The page table (memory map) indicates if the page is on disk or memory.

**Demand Paging uses a memory as a cache for the disk.**
When to load a page?

- Difficult to get right due to branches in code.
- Errors may result in removing useful pages.
- If the OS is wrong, page fault.
- Allowing more overlap of CPU and I/O if the OS guesses correctly.

**Pre-pageing:** OS guesses in advance which pages the process will need and pre-loads them into memory.

**Demand pageing:** OS loads a page the first time it is referenced.

- Page fault: interrupt that occurs when an instruction references a page that is not in memory.
- Process must give up the CPU while the page is being loaded.
- May remove a page from memory to make room for the new page.

When it is thought to be a page.

**Request pageing**: process tells an OS before it needs a page, and then...

- Difficult to do and is error-prone.
- Allows virtual address space to be larger than physical address space.

**Overlays**:
application programmer indicates when to load and remove the physical memory.

**At process start time**: the virtual address space must be no larger than...
Page table must be more sophisticated so that it knows where to find a

- Swap space
- Physical memory
- The system

At any given time, a page of virtual memory might exist in one or more of:

- Memory
  - Swap space: A portion of the disk is reserved for storing pages that are evicted from
    process it belongs to refers to it again.
  - If the page contains data, we need to save the data so that it can be reloaded if the
    disk.
  - If the page contains code, we could simply remove it since it can be re-loaded from

What happens when a page is removed from memory?

Swap Space

Implementation of Demand Paging
All of this is still functionally transparent to the user.

- If page is not in memory, OS picks a TLB entry to replace and fills it in as follows
  - invalidates TLB entry
- If page is in memory, OS picks a TLB entry to replace and then fills it in the new
  - invalidates TLB entry
- updated TLB entry

If the TLB hit rate is very high, use software to load the TLB.

In some implementations, the hardware loads the TLB on a TLB miss.

Updating the TLB

---

What value must $p$ have?

- If we want the effective access time to be only 10% slower than memory access time,
  \[ \text{Effective access time} = 1 \times (d - 1) + 200'000 \times (d - 1) \times 25 \text{ ms} \]
- If memory access time is 200 ns and a page fault takes 25 ms
  \[ \text{Effective access time} = 1 \times d + ma \times (d - 1) \times d \times \text{page fault time} \]

Let $d$ be the probability of a page fault ($0 < d < 1$).

---

Special locality refers to a process accessing an item in memory, it will tend to reference an adjacent item soon.

Temporal locality refers to a process accessing an item in memory, it will tend to reference the same item again soon.

Fortunately, processes typically exhibit locality of reference:

Theoretically, a process could access a new page with each instruction.

Performance of Demand Paging
transient
of the source and destination are in memory before starting the block

Solution: check that all pages between the starting and ending addresses

be undone.

Block transient instructions where the source and destination overlap can’t

Solution: unwind side effects

register 10.

- mov a, (r10)+: moves a into the address contained in register 10 and increments

What about instructions with side-effects (CISC)?

- the CPU state,

1. the faulting instruction,

2. Need hardware support to save

How does the OS transparently react to a faulting instruction?
Number of page faults?

Frame 3
Frame 2
Frame 1
A B C A B D A D B C B

FIFO: First-In-First-Out
Reference stream: A B C A B D A D B C B
Virtual pages: A B C D
3 page frames

Example: FIFO

LRU: Least Recently Used. Approximation of MIN that works well if the accessed page that is accessed farthest in the future (provably optimal [Belay96]). Problem? Being a frequency, it is simple to implement, but the OS can easily throw out a page that is the oldest page.

FIFO: First-In, First-Out. Throw out the oldest page.
Random: amazingly, this algorithm works pretty well.

Page Replacement Algorithms

on a page fault, we need to choose a page to evict
Number of page faults?

Frame 3
Frame 2
Frame 1
A | B | C | D | A | B | C | B

LRU: Least Recently Used. Throw out the page that has not been used in the longest time.

Example: LRU

Number of page faults?

Frame 3
Frame 2
Frame 1
A | B | C | D | A | B | C | B

MIN: Look into the future and throw out the page that will be accessed farthest in the future.

Example: MIN
of page frames.

With FIFO, the contents of memory can be completely different with a different number of page faults.

<table>
<thead>
<tr>
<th>Frame</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Frame 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Frame 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Frame 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FIFO:**

Does adding memory always reduce the number of page faults?

<table>
<thead>
<tr>
<th>Adding Memory</th>
</tr>
</thead>
</table>

When will LRU perform badly?

**Example: LRU**
and improve performance

A good page replacement algorithm can reduce the number of page faults

switch occurs.
- Processes can share memory more efficiently, reducing the costs when a context
  switch occurs.
- Processes can run without being fully loaded into memory.

Virtual address space can be larger than physical address space.

Benefits of demand paging:

Summary

Why?

With LRU, increasing the number of frames always decreases the number of page faults.

<table>
<thead>
<tr>
<th>Frame 4</th>
<th>Frame 3</th>
<th>Frame 2</th>
<th>Frame 1</th>
<th>Frame 3</th>
<th>Frame 2</th>
<th>Frame 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
</tbody>
</table>

LRU:

Adding Memory with LRU