Today: Demand Paged Virtual Memory

3. Allowing more processes than fit in memory to run concurrently.
2. A process to execute even if all of the process is not in memory, and
1. A process to be larger than physical memory, and

The illusion of an infinite virtual memory enables

much slower, cache or set of registers
2. analogous to the way in which main memory is a much larger, but
main memory

I 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

- Each program consists of a number of segments
- User view of programs

Segmentation

Last Class
### Demand Paged Virtual Memory: Example

**Key Ideas:**
- Local—The working set size of a process must fit in memory, and must stay there. (90/10 rule.)
- Else the effective memory access time will approach that of the disk 
- For efficiency reasons, memory access must reference pages that are in memory the vast majority of the time 
- 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.
- Difficult to get right due to branches in code.
- Errors may result in removing useful pages.
- If the OS is wrong — page fault!
- Allows more overlap of CPU and /O if the OS guesses correctly.

and pre-loads them into memory

**Pre-paging**: OS guesses in advance which pages the process will need

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

---

**When to load a page?**

When it is through with a page.

**Request paging**: 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.

**When the process starts**: The virtual address space must no larger then

---

**When to load a page?**
Page table must be more sophisticated so that it knows where to find a

- Swap space
- Physical memory
- The file 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 if it belongs to process and

- If the page contains data, we need to save the data in main memory so that it can be reloaded if the
  disk is.
- 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

7. Continuous failsafe process (why not continue current process?)
6. Updates the page table entry
5. Gets interrupt that page is loaded in memory
4. Context switches to another process where I/O is being
3. Starts loading new page into memory from disk
2. Invalidates the old page in the page table
1. Selects a page to replace (page replacement algorithm)

The OS checks that the address is valid. If so, it

If the page is not in memory, trap to the OS on first the reference

- In memory: not in memory (either on disk or bogus address)

Valid bit in page table indicates if page is in memory.

A copy of the entire program must be stored on disk. Why?
A 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**

**Performance of Demand Paging**

1. If this is still functionally transparent to the user.
   - If the page is not in memory, OS picks a TLB entry to replace and fills it in as follows
     - Invalidates TLB entry
     - Performs fault operations as described earlier
     - Updates TLB entry
     - Returns faulting process
   - If page is in memory, OS picks a TLB entry to replace and then fills it in the new
     - Valid bit in the TLB indicates if page is in memory
     - OS misses the TLB
     - TP on a TLB hit use the frame number to access memory
2. Write bit in the TLB indicates if page is in memory

What value must \( p \) have?

- If we want the effective access time to be only 10% slower than memory access time,
  \[
  \text{Effective access time} = (1 - 0.1) \times (d - 1) \times 200,000 + \frac{200,000 \times (d - 1) \times 25}{d - 1} + ma \times \text{page fault time}
  \]
- Let \( d \) be the probability of a page fault (0 \( \leq d \leq 1 \)).
Transfer

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 transfer instructions where the source and destination overlap can't

TransParent Page Faults

Solution: unwind side effects

register 10.

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

What about instructions with side-effects (CISC)

2. the CPU state.

1. the faulting instruction.

Need hardware support to save

How does the OS transparently restart a failing instruction?

TransParent Page Faults
Number of page faults?

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

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 farthest in the future (provably optimal [Belady66]). Problem?
MIN: (a.k.a. Opt) look into the future and throw out the page that will be
being accessed frequently.

FIFO: First-In, First-Out. Throw out the oldest page.

Random: amazingly, this algorithm works pretty well.

On a page fault, we need to choose a page to evict.

Page Replacement Algorithms
Number of page faults?

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

The longest time:

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

Example: LRU

Number of page faults?

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

Farthest in the future:

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

Example: MIN
With FIFO, the contents of memory can be completely different with a different number of page frames.

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

FIFO:

Does adding memory always reduce the number of page faults?

Example: LRU

When will LRU perform badly?
and improve performance

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

switch occurs.

- Processes can share memory more effectively, reducing the costs when a context

  - Processes start faster because they only need to load a few pages (for code and data)

  - 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.

**Adding Memory with LRU**