COS 318: Operating Systems
Virtual Memory Design Issues

Kai Li
Computer Science Department
Princeton University

(http://www.cs.princeton.edu/courses/cos318/)
Design Issues

- Thrashing and working set
- Backing store
- Simulate certain PTE bits
- Pin/lock pages
- Zero pages
- Shared pages
- Copy-on-write
- Distributed shared memory
- Virtual memory in Unix and Linux
- Virtual memory in Windows 2000
Virtual Memory Design Implications

- Revisit Design goals
  - Protection
    - Isolate faults among processes
  - Virtualization
    - Use disk to extend physical memory
    - Make virtualized memory user friendly (from 0 to high address)

- Implications
  - TLB overhead and TLB entry management
  - Paging between DRAM and disk

- VM access time
  
  \[ \text{Access time} = h \times \text{memory access time} + (1 - h) \times \text{disk access time} \]

  - E.g. Suppose memory access time = 100ns, disk access time = 10ms
    - If $h = 90\%$, VM access time is 1ms!
Thrashing

- Thrashing
  - Paging in and paging out all the time, I/O devices fully utilized
  - Processes block, waiting for pages to be fetched from disk

- Reasons
  - Processes require more physical memory than it has
  - Does not reuse memory well
  - Reuses memory, but it does not fit
  - Too many processes, even though they individually fit

- Solution: **working set** (last lecture)
  - Pages referenced by a process in the last T seconds
  - Two design questions
    - Which working set should be in memory?
    - How to allocate pages?
Working Set: Fit in Memory

- Maintain two groups
  - Active: working set loaded
  - Inactive: working set intentionally not loaded

- Two schedulers
  - A short-term scheduler schedules processes
  - A long-term scheduler decides which one active and which one inactive, such that active working sets fits in memory

- A key design point
  - How to decide which processes should be inactive
  - Typical method is to use a threshold on waiting time
Working Set: Global vs. Local Page Allocation

- The simplest is global allocation only
  - Pros: Pool sizes are adaptable
  - Cons: Too adaptable, little isolation (example?)

- A balanced allocation strategy
  - Each process has its own pool of pages
  - Paging allocates from its own pool and replaces from its own working set
  - Use a “slow” mechanism to change the allocations to each pool while providing isolation

- Design questions:
  - What is “slow?”
  - How big is each pool?
  - When to migrate?
Backing Store

- **Swap space**
  - When process is created, allocate a swap space for it
  - Need to load or copy executables to the swap space
  - Need to consider process space growth

- **Page creation**
  - Allocate a disk address?
  - What if the page never swaps out?
  - What if the page never gets modified?

- **Swap out**
  - Use the same disk address?
  - Allocate a new disk address?
  - Swap out one or multiple pages?

- **Text pages**
  - They are read only in most cases. Treat them differently?
Revisit Address Translation

- Map to page frame and disk
  - If valid bit = 1, map to pp# physical page number
  - If valid bit = 0, map to dp# disk page number

- Page out
  - Invalidate page table entry and TLB entry
  - Copy page to disk
  - Set disk page number in PTE

- Page in
  - Find an empty page frame (may trigger replacement)
  - Copy page from disk
  - Set page number in PTE and TLB entry and make them valid

```
subl $20 %esp
movl 8(%esp), %eax
```

Process

**TLB**

<table>
<thead>
<tr>
<th>v</th>
<th>vp#</th>
<th>pp#</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>vp#</td>
<td>pp#</td>
</tr>
<tr>
<td>i</td>
<td>vp#</td>
<td>dp#</td>
</tr>
<tr>
<td>v</td>
<td>vp#</td>
<td>pp#</td>
</tr>
<tr>
<td>v</td>
<td>vp#</td>
<td>pp#</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: x86 Paging Options

- **Flags**
  - PG flag (Bit 31 of CR0): enable page translation
  - PSE flag (Bit 4 of CR4): 0 for 4KB page size and 1 for large page size
  - PAE flag (Bit 5 of CR4): 0 for 2MB pages when PSE = 1 and 1 for 4MB pages when PSE = 1, extending physical address space to 36 bit

- 2MB and 4MB pages are mapped directly from directory entries
- 4KB and 4MB pages can be mixed

![Page-Table Entry (4-KByte Page)](image)
Example: x86 Directory Entry

**Page-Directory Entry (4-KByte Page Table)**

<table>
<thead>
<tr>
<th>31</th>
<th>12</th>
<th>11</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page-Table Base Address</td>
<td>Avail</td>
<td>G</td>
<td>P</td>
<td>S</td>
<td>0</td>
<td>A</td>
<td>P</td>
<td>C</td>
<td>D</td>
<td>P</td>
<td>W</td>
<td>T</td>
</tr>
</tbody>
</table>

- Available for system programmer’s use
- Global page (Ignored)
- Page size (0 indicates 4 KBytes)
- Reserved (set to 0)
- Accessed
- Cache disabled
- Write-through
- User/Supervisor
- Read/Write
- Present

**Page-Directory Entry (4-MByte Page)**

| 31 | 22 | 21 | 13 | 12 | 11 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|---|

- Page Table Attribute Index
- Available for system programmer’s use
- Global page
- Page size (1 indicates 4 MBytes)
- Dirty
- Accessed
- Cache disabled
- Write-through
- User/Supervisor
- Read/Write
- Present
Simulating PTE Bits

- **Simulating modify bit using read/write bit**
  - Set pages read-only if they are read-write
  - Use a reserved bit to remember if the page is really read-only
  - On a write fault
    - If it is not really read-only, then record a modify in the data structure and change it to read-write
    - Restart the instruction

- **Simulating access (reference) bit using valid bit**
  - Invalidate all valid bits (even they are valid)
  - Use a reserved bit to remember if a page is really valid
  - On a page fault
    - If it is a valid reference, set the valid bit and place the page in the LRU list
    - If it is an invalid reference, do the page replacement
    - Restart the faulting instruction
Pin (or Lock) Page Frames

- **When do you need it?**
  - When DMA is in progress, you don’t want to page the pages out to avoid CPU from overwriting the pages

- **How to design the mechanism?**
  - A data structure to remember all pinned pages
  - Paging algorithm checks the data structure to decide on page replacement
  - Special calls to pin and unpin certain pages

- **How would you implement the pin/unpin calls?**
  - If the entire kernel is in physical memory, do we still need these calls?
Zero Pages

- Zeroing pages
  - Initialize pages with 0’s
  - Heap and static data are initialized

- How to implement?
  - On the first page fault on a data page or stack page, zero it
  - Have a special thread zeroing pages

- Can you get away without zeroing pages?
Shared Pages

- PTEs from two processes share the same physical pages
  - What use cases?
- APIs
  - Shared memory calls
- Implementation issues
  - Destroy a process with shared pages
  - Page in, page out shared pages
  - Pin and unpin shared pages
  - Derive the working set for a process with shared pages
Copy-On-Write

- A technique to avoid prepare all pages to run a large process

- Method
  - Child’s address space uses the same mapping as parent’s
  - Make all pages read-only
  - Make child process ready
  - On a read, nothing happens
  - On a write, generates a fault
    - map to a new page frame
    - copy the page over
    - restart the instruction

- Issues
  - How to destroy an address space?
  - How to page in and page out?
  - How to pin and unpin?
Distributed Shared Memory

- Run shared memory program on a cluster of computers

- Method
  - Multiple address space mapped to “shared virtual memory”
  - Page access bits are set according to coherence rules
    - Exclusive writer
    - N readers
  - A read fault will invalidate the writer, make read only and copy the page
  - A write fault will invalidate another writer or all readers and copy page

- Issues
  - Thrashing
  - Copy page overhead
  - Synchronization
Address Space in Unix

- **Stack**
- **Data**
  - Un-initialized: BSS (Block Started by Symbol)
  - Initialized
  - \texttt{brk(addr)} to grow or shrink
- **Text**: read-only
- **Mapped files**
  - Map a file in memory
  - \texttt{mmap(addr, len, prot, flags, fd, offset)}
  - \texttt{unmap(addr, len)}
Virtual Memory in BSD4

- **Physical memory partition**
  - Core map (pinned): everything about page frames
  - Kernel (pinned): the rest of the kernel memory
  - Frames: for user processes

- **Page replacement**
  - Run page daemon until there is enough free pages
  - Early BSD used the basic Clock (FIFO with 2nd chance)
  - Later BSD used Two-handed Clock algorithm
  - Swapper runs if page daemon can’t get enough free pages
    - Looks for processes idling for 20 seconds or more
    - 4 largest processes
    - Check when a process should be swapped in
Virtual Memory in Linux

- Linux address space for 32-bit machines
  - 3GB user space
  - 1GB kernel (invisible at user level)

- Backing store
  - Text segment uses executable binary file as backing storage
  - Other segments get backing storage on demand

- Copy-on-write for forking off processes

- Multi-level paging
  - Directory, middle (nil for Pentium), page, offset
  - Kernel is pinned
  - Buddy algorithm with carving slabs for page frame allocation

- Replacement
  - Keep certain number of pages free
  - Clock algorithm on paging cache and file buffer cache
  - Clock algorithm on unused shared pages
  - Modified Clock on memory of user processes (most physical pages first)
Address Space in Windows 2K/XP

- **Win2k user address space**
  - Upper 2GB for kernel (shared)
  - Lower 2GB – 256MB are for user code and data (Advanced server uses 3GB instead)
  - The 256MB contains for system data (counters and stats) for user to read
  - 64KB guard at both ends

- **Virtual pages**
  - Page size
    - 4KB for x86
    - 8 or 16KB for IA64
  - States
    - Free: not in use and cause a fault
    - Committed: mapped and in use
    - Reserved: not mapped but allocated
Backing Store in Windows 2K/XP

- **Backing store allocation**
  - Win2k delays backing store page assignments until paging out
  - There are up to 16 paging files, each with an initial and max sizes

- **Memory mapped files**
  - Delayed write back
  - Multiple processes can share mapped files w/ different accesses
  - Implement copy-on-write
Paging in Windows 2K/XP

- Each process has a working set with
  - Min size with initial value of 20-50 pages
  - Max size with initial value of 45-345 pages

- On a page fault
  - If working set < min, add a page to the working set
  - If working set > max, replace a page from the working set

- If a process has a lot of paging activities, increase its max

- Working set manager maintains a large number of free pages
  - In the order of process size and idle time
  - If working set < min, do nothing
  - Otherwise, page out the pages with highest “non-reference” counters in a working set for uniprocessors
  - Page out the oldest pages in a working set for multiprocessors

- The last 512 pages are never taken for paging
More Paging in Windows 2K/XP

Diagram:

- **Working Sets**
- **Modified page list**
- **Modified page writer**
- **Standby page list**
- **Dealloc**
- **Free page list**
- **Zero page thread**
- **Zeroed page list**
- **Bad page list**

- **Page in**
- **Zero paging**
- **Soft fault**
- **Replaced**
- **Process exit**
Summary

- Must consider many issues
  - Global and local replacement strategies
  - Management of backing store
  - Primitive operations
    - Pin/lock pages
    - Zero pages
    - Shared pages
    - Copy-on-write

- Shared virtual memory can be implemented using access bits
- Real system designs are complex