Chapter 9: Virtual Memory

张竞慧
办公室：计算机楼366室
电邮：jhzhang@seu.edu.cn
主页：http://cse.seu.edu.cn/PersonalPage/zjh/
电话：025-52091017
Chapter 9: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples
Background

- **Virtual memory** – separation of user logical memory from physical memory.
  - Only part of the program needs to be in memory for execution.
  - Logical address space can therefore be much larger than physical address space.
  - More programs can be run at the same time
  - Less I/O be needed to load or swap

- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation
Virtual Memory That is Larger Than Physical Memory
Chapter 9: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples
Demand Paging

- Bring a page into memory only when it is needed.
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users

- Page is needed $\Rightarrow$ reference to it
  - invalid reference $\Rightarrow$ abort
  - not-in-memory $\Rightarrow$ bring to memory
Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated
  (1 ⇒ in-memory, 0 ⇒ not-in-memory)
- Initially valid–invalid bit is set to 0 on all entries.
- During address translation, if valid–invalid bit in page table entry is 0 ⇒ page fault.
Page Table When Some Pages Are Not in Main Memory

Logical memory:

- A
- B
- C
- D
- E
- F
- G
- H

Page table:

- Frame 0: Page 4, Valid (v)
- Frame 1: Page 6, Invalid (i)
- Frame 2: Page 9, Valid (v)
- Frame 3: Page i
- Frame 4: Page i
- Frame 5: Page i
- Frame 6: Page i
- Frame 7: Page i

Physical memory:

- Frame 0: Page A
- Frame 1: Page B
- Frame 2: Page C
- Frame 3: Page D
- Frame 4: Page E
- Frame 5: Page F
- Frame 6: Page G
- Frame 7: Page H

Valid-invalid bit:

- Frame 0: Valid
- Frame 1: Invalid
- Frame 2: Valid
- Frame 3: Invalid
- Frame 4: Invalid
- Frame 5: Invalid
- Frame 6: Invalid
- Frame 7: Invalid
Page Fault

- If there is ever a reference to a page, first reference will trap to OS ⇒ page fault

- OS looks at page table to decide:
  - Invalid reference ⇒ abort.
  - Just not in memory.

- Get empty frame.

- Swap page into frame.

- Reset tables, validation bit = 1.

- Restart instruction
  - block move
  - auto increment/decrement location
Steps in Handling a Page Fault

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

Load M

Operating System Concepts
Performance of Demand Paging

- Extreme case – start process with no pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - And for every other process pages on first access
  - Pure demand paging
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
Performance of Demand Paging

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = \((1 - p) \times 200 + p \times 8\) milliseconds
  \[
  = 200 + p \times 7,999,800
  \]
- 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!!
Performance of Demand Paging

- If want performance degradation < 10 percent
  \[ 220 > 200 + 7,999,800 \times p \]
  \[ 20 > 7,999,800 \times p \]
  \[ p < 0.0000025 \]
- < one page fault in every 400,000 memory accesses
Chapter 9: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples
Process Creation

Virtual memory allows other benefits during process creation:

- Copy-on-Write

- Memory-Mapped Files (Later)
Copy-on-Write

- Copy-on-Write (COW) allows both parent and child processes to initially *share* the same pages in memory. (No demand paging)
  - If either process modifies a shared page, only then is the page copied.
- COW allows more efficient process creation as only modified pages are copied.
Before Process 1 Modifies Page C

- process\(_1\) connects to page A in physical memory.
- process\(_1\) connects to page B in physical memory.
- process\(_1\) connects to page C in physical memory.
- process\(_2\) connects to page A in physical memory.
After Process 1 Modifies Page C

process₁

physical memory

page A

page B

page C

Copy of page C

process₂
Chapter 9: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples
What happens if there is no free frame?

- Page replacement – find some page in memory, but not really in use, swap it out
  - algorithm
  - performance – want an algorithm which will result in minimum number of page faults

- Same page may be brought into memory several times
Page Replacement

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

Logical memory for user 1:
- Frame 0: H
- Frame 1: load M
- Frame 2: J
- Frame 3: M

Page table for user 1:
- Page 0: valid
- Page 1: valid
- Page 2: valid
- Page 3: invalid

Logical memory for user 2:
- Frame 0: A
- Frame 1: B
- Frame 2: D
- Frame 3: E

Page table for user 2:
- Page 4: valid
- Page 5: valid
- Page 6: valid
- Page 7: valid
Basic Page Replacement

1. Find the location of the desired page on disk.

2. Find a free frame:
   - If there is a free frame, use it.
   - If there is no free frame, use a page replacement algorithm to select a victim frame.

3. Read the desired page into the (newly) free frame. Update the page and frame tables.

4. Restart the instruction.
Page Replacement

1. swap out victim page
2. change to invalid
3. swap desired page in
4. reset page table for new page
Page Replacement Algorithms

- 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.
- In all our examples, the reference string is 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.
Graph of Page Faults Versus The Number of Frames

The graph shows the relationship between the number of page faults and the number of frames. As the number of frames increases, the number of page faults decreases exponentially.
First-In-First-Out (FIFO) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)

1 | 1 | 4 | 5
2 | 2 | 1 | 3 | 9 page faults
3 | 3 | 2 | 4
First-In-First-Out (FIFO) Algorithm

- 4 frames

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>10 page faults</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

- FIFO Replacement – Belady’s Anomaly
  - more frames ⇒ more page faults
FIFO Illustrating Belady’s Anomaly
FIFO Page Replacement

reference string
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

page frames

7 7 7 2
0 0 0
1 1

2 2 4 4 4 0
3 3 3 2 2 2
1 1

0 0
3 2

1 0 0
2 2 1
Optimal Algorithm

- Replace page that will not be used for longest period of time.
- 4 frames example

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

1  
2  
3  
4  
5  

6 page faults

- How do you know this?
- Used for measuring how well your algorithm performs.
Optimal Page Replacement

| reference string | 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |
| page frames      | 7 | 7 | 7 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 7 |
|                  | 0 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 0 |
|                  | 1 | 1 | 3 | 3 | 3 | 1 | 1 | 1 | 1 | 1 | 1 |
Least Recently Used (LRU) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

- Counter implementation (how?)
  - Every page entry has a counter; every time a page is referenced through this entry, copy the clock into the counter.
  - When a page needs to be changed, look at the counters to determine which are to change.
LRU Page Replacement

reference string
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

page frames

7 7 7 2 2
0 0 0 0
1 1 1 3

4 4 4 0
0 0 3 3
3 2 2 2

1 1 1
3 0 0
2 2 7
LRU Algorithm (Cont.)

- Stack implementation – keep a stack of page numbers in a double link form: (How?Exercise1)
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - No search for replacement
Use Of A Stack to Record The Most Recent Page References

reference string

4 7 0 7 1 0 1 2 1 2 7 1 2

stack before a
2
1
0
7
4

stack after b
7
2
1
0
4

a
b
LRU Approximation Algorithms

- Reference bit
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1.
  - Replace the one which is 0 (if one exists). We do not know the order, however.

- Additional-Reference-Bits Algorithm
  - Keep an 8-bit bytes for each page
  - At regular intervals shifts the bits right 1 bit, shift the reference bit into the high-order bit
  - Interpret these 8-bit bytes as unsigned intergers, the page with lowest number is the LRU page
LRU Approximation Algorithms (Cont.)

- Second chance (linklist approach?)
  - Need reference bit.
  - Clock replacement.
  - If page to be replaced (in clock order) has reference bit = 1. then:
    - set reference bit 0.
    - leave page in memory.
    - replace next page (in clock order), subject to same rules.
Second-Chance (clock) Page- Replacement Algorithm

circular queue of pages

(a)

next victim

reference bits

pages

0

0

1

0

1

0

...

1

1

reference bits

pages

0

0

1

0

1

0

...

1

1

(b)
Counting Algorithms

- Keep a counter of the number of references that have been made to each page.

- LFU Algorithm: replaces page with smallest count.

- MFU Algorithm: based on the argument that the page with the smallest count was probably just brought in and has yet to be used.
Chapter 9: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples
Allocation of Frames

- Each process needs **minimum** number of pages. *(why? Restart instruction)*

- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
  - Instruction is 6 bytes, might span 2 pages.
  - 2 pages to handle **from**.
  - 2 pages to handle **to**.

- Two major allocation schemes.
  - Fixed allocation
  - Priority allocation
Fixed Allocation

- Equal allocation – e.g., if 100 frames and 5 processes, give each 20 pages.
- Proportional allocation – Allocate according to the size of process.

\[
s_i = \text{size of process } p_i \\
S = \sum s_i \\
m = \text{total number of frames} \\
a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m
\]
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.
  - 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. (benefit? Weakness?)

- **Local** replacement – each process selects from only its own set of allocated frames.
Chapter 9: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples
Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high. This leads to: (how it happens?)
  - low CPU utilization.
  - operating system thinks that it needs to increase the degree of multiprogramming.
  - another process added to the system.

**Thrashing** ≡ a process is busy swapping pages in and out.
Why does paging work? (how to know the frame number to allocate to a process?)
Locality model
- Process migrates from one locality to another.
- Localities may overlap.

Why does thrashing occur?
\[ \Sigma \text{size of locality} > \text{total memory size} \]
Locality In A Memory-Reference Pattern
Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instruction

- $WSS_i$ (working set of Process $P_i$) = total number of pages referenced in the most recent $\Delta$ (varies in time)
  - if $\Delta$ too small will not encompass entire locality.
  - if $\Delta$ too large will encompass several localities.
  - if $\Delta = \infty \implies$ will encompass entire program.
Working-Set Model (Cont.)

\[ D = \sum WSS_i \equiv \text{total demand frames} \]

- if \( D > m \) \( \Rightarrow \) Thrashing

- Policy if \( D > m \), then suspend one of the processes.
Working-set model

Page reference table

\[ \ldots 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 \ldots \]

\[ \Delta \]

\[ t_1 \]

\[ WS(t_1) = \{1, 2, 5, 6, 7\} \]

\[ \Delta \]

\[ t_2 \]

\[ WS(t_2) = \{3, 4\} \]
Keeping Track of the Working Set

- Approximate with interval timer + a reference bit

- Example: $\Delta = 10,000$
  
  - Timer interrupts after every 5000 time units.
  
  - Keep in memory 2 bits for each page.
  
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0.
  
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set.

Why is this not completely accurate?

- Improvement = 10 bits and interrupt every 1000 time units.

What happens when page fault occurs?
- Establish “acceptable” page-fault rate.
  - If actual rate too low, process loses frame.
  - If actual rate too high, process gains frame.
Chapter 9: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples
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.
- 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 file access by treating file I/O through memory rather than `read()` `write()` system calls.
- Also allows several processes to map the same file allowing the pages in memory to be shared.
Memory Mapped Files

process A
virtual memory

physical memory

process B
virtual memory

disk file
Memory-Mapped Shared Memory in Windows

[Diagram showing memory-mapped shared memory between two processes.]
Example: linux mmap

- Following link:
Chapter 9: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples
Allocating Kernel Memory

- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes, fragmentation need to be taken care of
  - Some kernel memory needs to be contiguous
Buddy System

- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using **power-of-2 allocator**
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2
  - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
    - Continue until appropriate sized chunk available
Buddy System Allocator

21KB kernel memory allocation request

Fragmentation problem
Slab Allocator

- Alternate strategy
- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure
  - Each cache filled with **objects** – instantiations of the data structure
Slab Allocator (Cont)

- When cache created, filled with objects marked as **free**
- When structures stored, objects marked as **used**
- If slab is full of used objects, next object allocated from empty slab
  - If no empty slabs, new slab allocated
- Benefits include no fragmentation, fast memory request satisfaction
Slab Allocation

- **Kernel objects**
- **Caches**
- **Slabs**

- 3 KB objects
- 7 KB objects

Physical contiguous pages
Slab Allocation

- For example process descriptor is of type struct task_struct
  - Approx 1.7KB of memory
  - New task -> allocate new struct from cache
  - Will use existing free struct task_struct

- Slab can be in three possible states
  - Full – all used
  - Empty – all free
  - Partial – mix of free and used

- Upon request, slab allocator
  - Uses free struct in partial slab
  - If none, takes one from empty slab
  - If no empty slab, create new empty
Chapter 9: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples
Other Issues -- 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 (work set model)
- But if prepaged pages are unused, I/O and memory was wasted

Assume $s$ pages are prepaged and $\alpha$ of the pages is used

- Is cost of $s \cdot \alpha$ save pages faults $>$ or $<$ than the cost of prepaging $s \cdot (1 - \alpha)$ unnecessary pages?
- $\alpha$ near zero $\Rightarrow$ prepaging loses
Other Issues – Page Size

Page size selection must take into consideration:

- Fragmentation(?)
- Table size (?)
- I/O overhead(?)
- Locality(?)
Other 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
- Increase the Page Size
- Provide Multiple Page Sizes
  - This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation
Other Issues – Program Structure

- Program structure
  - int A[][] = new int[1024][1024];
  - Each row is stored in one page
  - Program 1
    ```java
    for (j = 0; j < A.length; j++)
        for (i = 0; i < A.length; i++)
            A[i, j] = 0;
    ```
    1024 x 1024 page faults
  - Program 2
    ```java
    for (i = 0; i < A.length; i++)
        for (j = 0; j < A.length; j++)
            A[i, j] = 0;
    ```
    1024 page faults
Chapter 9: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples
Operating System Examples

- Windows XP
- Solaris
Windows XP

- Uses demand paging with clustering. Clustering brings in pages surrounding the faulting page.
- Processes are assigned working set minimum and working set maximum.
- Working set minimum is the minimum number of pages the process is guaranteed to have in memory.
Windows XP (Cont)

- A process may be assigned as many pages up to its working set maximum.
- When the amount of free memory in the system falls below a threshold, **automatic working set trimming** is performed to restore the amount of free memory.
- Working set trimming removes pages from processes that have pages in excess of their working set minimum.
Solaris

- Maintains a list of free pages to assign faulting processes
- *Lotsfree* – threshold parameter (amount of free memory) to begin paging
- *Desfree* – threshold parameter to increasing paging
- *Minfree* – threshold parameter to being swapping
Solaris (Cont)

- Paging is performed by *pageout* process.
- Pageout scans pages using modified clock algorithm.
- *Scanrate* is the rate at which pages are scanned. This ranges from *slowscan* to *fastscan*.
- Pageout is called more frequently depending upon the amount of free memory available.
Solaris 2 Page Scanner

- 8192 fastscan
- 100 slowscan

Scan rate vs. amount of free memory:
- minfree
- desfree
- lotsfree