Chapter 9: Virtual-Memory Management
Chapter 9: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocation Kernel Memory
- Other Consideration
- 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.
  - Allows address spaces to be shared by several processes.
  - Allows for more efficient process creation.

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

- stack
- heap
- data
- code

Max

0
Shared Library Using Virtual Memory

- Stack
- Shared library
- Heap
- Data
- Code

- Stack
- Shared library
- Heap
- Data
- Code

Shared pages
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
Transfer of a Paged Memory to Contiguous Disk Space

Diagram showing the transfer of memory pages between main memory and disk space. Program A is swapped out, freeing pages 0 to 3 for program B. Program B is swapped in, occupying pages 4 to 7. The sequence of page swap operations is illustrated.
Page Table When Some Pages Are Not in Main Memory
Steps in Handling a Page Fault

1. Trap

2. Page is on backing store

3. Operating system

4. Bring in missing page

5. Reset page table

6. Restart instruction

Load M
What happens if there is no free frame?

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

- Same page may be brought into memory several times.
Software Support

- Able to restart any instruction after a page fault
- Difficulty: when one instruction modifies several different locations
  
  e.g., IBM 390/370 MVC move block2 to block1

Solutions
1. Access both ends of both blocks before moving
2. Use temporary registers to hold the values of overwritten locations – for the undo
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 \times (\text{page fault overhead} 
  + [\text{swap page out}] 
  + \text{swap page in} 
  + \text{restart overhead})
  \]
Page Fault Processing: details

1. Trap to the OS
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location on the disk
5. Issue a read from the disk to a free frame:
   a. Wait in a queue for this device until the read request is serviced
   b. Wait for the device seek and/or latency time
   c. Begin the transfer of the page to a free frame
Page Fault Processing: details

6. While waiting, allocate the CPU to some other user (CPU scheduling)
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user (if step 6 is executed)
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show that the desired page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction
Copy-on-Write

- Copy-on-Write (COW) allows both parent and child processes to initially *share* the same pages in memory.
- 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.
- Free pages are allocated from a *pool* of zeroed-out pages.
vfork (): virtual memory fork

- vfork(): without COW capability
- fork(): with COW capability

- With vfork(), the parent process is suspended, and the child process uses the address space of the parent
- vfork() is intended to be used when the child process calls exec() immediately after creation
- Because no copying of pages takes place, vfork() is an extremely efficient method of process creation
Before Process 1 Modifies Page C
After Process 1 Modifies Page C
Page Replacement

- When a page fault occurs with no free frame
  - swap out a process, freeing all its frames, or
  - **page replacement**: find one not currently used and free it.
    - 😞: two page transfers
    - Solution: modify bit (dirty bit)
- Solve **two major problems** for demand paging
  - frame-allocation algorithm:
    - how many frames to allocate to a process
  - page-replacement algorithm:
    - select the frame to be replaced
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

logical memory for user 2

page table for user 1

page table for user 2

valid-invalid bit

monitor

physical memory
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 process
Page Replacement

1. Swap out victim page
2. Change to invalid
3. Swap desired page in
4. Reset page table for new page

physical memory

frame valid-invalid bit

page table
Page Replacement Algorithms

- **Goal**: 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
- For example, the reference string is 1, 4, 1, 2, 1, 3, 5, 1, 2, 3, 4, 5
Graph of Page Faults Versus The Number of Frames
Page Replacement Algorithms

- FIFO algorithm
- Optimal algorithm
- LRU algorithm
- LRU approximation algorithms
  - additional-reference-bits algorithm
  - second-chance algorithm
  - enhanced second-chance algorithm
- Counting algorithm
  - LFU
  - MFU
- Page buffering algorithm
The FIFO Algorithm

- Simplest
- Performance is not always good

- **Page out** a sequence of **active pages**
  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

- **Belady's anomaly**

allocated frames $\uparrow \Rightarrow$ page-fault rate $\uparrow$
FIFO Page Replacement

<table>
<thead>
<tr>
<th>Reference String</th>
<th>Page Frames</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1</td>
<td>7 7 7 2 7 2 2 4 3 3 3 2 2 2 1 1 1 1 0 0 3 3 3 2 2 2 1</td>
</tr>
</tbody>
</table>

FIFO Page Replacement

Silberschatz, Galvin and Gagne ©2005
Optimal Algorithm

- Has the lowest page-fault rate of all algorithms
- It replaces the page that will not be used for the longest period of time.
- difficult to implement, because it requires future knowledge
- used mainly for comparison studies
LRU Algorithm (Least Recently Used)

- An approximation of optimal algorithm:
  - looking backward, rather than forward.
- It replaces the page that has not been used for the longest period of time.
- It is often used, and is considered as quite good.
Two Implementations

counter (clock):

- time-of-used field for each page table entry

⚠️: 1. write counter to the field for each access
    2. search for the LRU

Stack: a stack of page number

- move the reference page form middle to the top
- best implemented by a doubly linked list

😊: no search

🤔: change six pointers per reference at most

```
2
1
0
7
4

reference 7
```

```
7
2
1
0
4

Head

Tail
```
Stack Algorithm

A property of algorithms

- **Stack algorithm**: the set of pages in memory for \( n \) frames is always a subset of the set of pages that would be in memory with \( n + 1 \) frames.

- Stack algorithms do not suffer from Belady's anomaly.

- Both *optimal* algorithm and **LRU** algorithm are stack algorithm.

- Few systems provide sufficient hardware support for the LRU page-replacement.

  ⇒ **LRU approximation algorithms**
LRU Approximation Algorithms

- Reference bit: Initially, all bits are cleared.
- When a page is referenced, its reference bit is set by hardware.
- Do the above process in a fixed period.
- We do not know the order of use, but we know which pages were used and which were not used.
Additional-reference-bits Algorithm

- Keep a $k$-bit byte for each page in memory
- At regular intervals,
  - shift right the $k$-bit (discarding the lowest)
  - copy reference bit to the highest
- Replace the page with smallest number (byte)
  - if not unique, FIFO or replace all
**(k=8)**

```
history ×

11010111
00110011
10100000
00001111
00100001
10000000
00000001
```

```
+ history

11101011
00110011
11010000
10000111
00100000
10000000
10000000
```

Every 100 ms, a timer interrupt transfers control to OS.
Second-Chance (clock) Page-Replacement Algorithm

Check pages in FIFO order (circular queue). If reference bit = 0, replace it else set to 0 and check next.
Enhanced Second Chance Algorithm

- Consider the pair (reference bit, modify bit), categorized into four classes:
  - (0,0): neither used and dirty
  - (0,1): not used but dirty
  - (1,0): used but clean
  - (1,1): used and dirty

- The algorithm: replace the first page in the lowest nonempty class

- 😞: search time
- 😊: reduce I/O (for swap out)
Counting Algorithms

- **LFU Algorithm** (least frequently used)
  - keep a counter for each page
  - Idea: An actively used page should have a large reference count.
  - Used heavily $\rightarrow$ large counter $\rightarrow$ may no longer needed but in memory

- **MFU Algorithm** (most frequently used)
  - Idea: The page with the smallest count was probably just brought in and has yet to be used.

- Both counting algorithms are not common
  - implementation is expensive
  - do not approximate OPT algorithm very well
Page Buffering Algorithms

- Keep a pool of free frames
  - the desired page is read before the victim is written out
  - allows the process to restart as soon as possible

- Maintain a list of modified pages (expansion)
  - When paging device is idle, a modified page is written to the disk and its modify bit is reset.

- Keep a pool of free frames but to remember which page was in each frame
  - possible to reuse an old page
Allocation of Frames

- Each process needs *minimum* number of pages (why?)
- 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.
  - $s_i = \text{size of process } p_i$
  - $S = \sum s_i$
  - $m = \text{total number of frames}$
  - $a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$

\[
m = 64
\]
\[
s_1 = 10
\]
\[
s_2 = 127
\]
\[
a_1 = \frac{10}{137} \times 64 \approx 5
\]
\[
a_2 = \frac{127}{137} \times 64 \approx 59
\]
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.
  - e.g., allow a high-priority process to take frames from a low-priority process
  - good system performance and thus is common used

- **Local** replacement – each process selects from only its own set of allocated frames.
Thrashing (1)

- If allocated frames < minimum number
  \[ \Rightarrow \text{Very high paging activity} \]
- A process is **thrashing** if it is **spending more time paging** than executing.

\[ \]
Thrashing (2)

- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  - 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
Locality In A Memory-Reference Pattern
Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references

- $WSS_i$ (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 $\Delta = \infty \Rightarrow$ will encompass entire program

- $D = \Sigma WSS_i \equiv$ total demand frames

- if $D > m \Rightarrow$ Thrashing

- Policy if $D > m$, then suspend one of the processes
Working-set model

😊: 1. Prevent thrashing while keeping the degree of multiprogramming as high as possible.
2. Optimize CPU utilization
😊: too expensive for tracking

Page reference table

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

\[ W_{S(t_1)} = \{1, 2, 5, 6, 7\} \]
\[ W_{S(t_2)} = \{3, 4\} \]
Approximate working set by using a fixed interval timer interrupt and a reference bit

- $\Delta = 10,000$ references, a timer interrupt every 5000 references, 2-bit history
  - copy and clear the reference bit for each interrupt
  - In case of page fault, a page is referenced within last 10,000 to 15,000 references can be identified

<table>
<thead>
<tr>
<th>time</th>
<th>0</th>
<th>~ 5,000</th>
<th>~ 10,000~</th>
</tr>
</thead>
<tbody>
<tr>
<td>reference</td>
<td>P1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>bits</td>
<td>P2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>P3</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

$\Delta = 10,000$

WS=$\{P1, P3\}$
Page Fault Frequency Scheme

- The knowledge of the working set can be useful for prepaging, but it seems a rather clumsy way to control thrashing.

- **Page fault frequency directly** measures and controls the page-fault rate to prevent thrashing.
  - Establish **upper and lower bounds** on the desired page-fault rate of a process.
  - If page fault rate exceeds the upper limit
    - allocate the process another frame
  - If page fault rate falls below the lower limit
    - remove the process a frame
Page-Fault Frequency Scheme

- Establish “acceptable” page-fault rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame
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
Allocating Kernel Memory

- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
  - 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

Physically contiguous pages

256 KB

128 KB

\(^{A_L}\)

128 KB

\(^{A_R}\)

64 KB

\(^{B_L}\)

64 KB

\(^{B_R}\)

A request of 23 KB

32 KB

\(^{C_L}\)

32 KB

\(^{C_R}\)
Slab Allocator

- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure (semaphores, process descriptors, file objects, ...)
  - Each cache filled with **objects** – instantiations of the data structure
- 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
Other Considerations

- Prepaging
- Page size selection
  - Fragmentation
  - Table size
  - I/O overhead
  - Locality
- Inverted page table
- Program structure
- I/O interlock
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
- 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 $s \alpha$ save pages faults $>$ or $<$ than the cost of prepaging
    - $s \alpha (1 - \alpha)$ unnecessary pages?
  - $\alpha$ near zero $\Rightarrow$ prepaging loses
Page size

usually, $2^{12}(4K) \sim 2^{22} (4M)$ size

- memory utilization (small internal fragmentation)  
  $\Rightarrow$ small size

- minimize I/O time (less seek, latency)  
  $\Rightarrow$ large size

- reduce total I/O (improve locality) $\Rightarrow$ small size  
  better resolution, allowing us to isolate only the memory that is actually needed.

- minimize number of page faults $\Rightarrow$ large size

Trend: larger

- CPU speed/memory capacity increase faster than disks. Page faults are more costly today.
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. This may lead to an increase in fragmentation as not all applications require a large page size

- Provide Multiple Page Sizes. This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation.
Inverted Page Table

- Reduce the amount of physical memory that is needed to track virtual-to-physical address translations. `<pid, page#>`

- The table no longer contains complete information about the logical address of a process and that information is required if a referenced page is not currently in memory.

- Demand paging requires this to process page faults. An external page table (one per process) must be kept.

- Do external page tables negate the utility of inverted page tables?
  - They do not need to be available quickly → paged in and out memory as necessary → Another page fault may occur as it pages in the external page table
Inverted Page Table Architecture

CPU

logical address

physical address

physical memory

page table

search

pid  p  d

i  d

pid  p
Careful selection of data/programming structure can increase locality

```pascal
var A: array[1..128, 1..128] of integer;
    for j := 1 to 128 do 
        for i := 1 to 128 do 
            /* 128 x 128 = 16,384 page faults */  
            A[i,j] := 0;

/* 128 page faults */  
A[i,j] := 0;
```

- **Stack is better than hash**
  - Stack: good locality since access is always made to the top
  - Hash: bad locality since designed to scatter references
I/O Interlock

Sometimes, we need to allow some of the pages to be locked in memory

- An example
  1. Process A prepare a page as I/O buffer and then waiting for an I/O device
  2. Process B takes the frame of A’s I/O page
  3. I/O device ready for A, a page fault occurs

- Solutions:
  - Never execute I/O to user memory
    (system memory ⇔ I/O device)
  - Allow pages to be locked (using a lock bit)
Real-time processing

- Virtual memory introduces unexpected, long delay
- Thus, real time system almost never have virtual memory
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
- 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
- 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
Homework

- 4, 8, 10, 12
- Due: 26, Dec.
End of Chapter 9