Chapter 9: Virtual Memory
Chapter 9: Virtual Memory

- Background
- Demand Paging
- Process Creation
- Page Replacement
- Allocation of Frames
- Thrashing
- Demand Segmentation
- Operating System Examples
Background

- **Virtual memory** – separation of user logical memory from physical memory.
  - **Only part of a program** needs to be in memory for execution (can really execute only one instruction at a time)
    - Only have to load code that is needed
    - Less I/O, so potential performance gain
    - More programs in memory, so better resource allocation and throughput
  - **Logical address space** can therefore be much larger than physical address space.
    - Programs can be larger than memory
    - Less management required by programmers
  - Need to allow **pages to be swapped in and out**

- **Virtual memory** can be implemented via:
  - **Demand paging** (for both paging and swapping)
  - **Demand segmentation**
Virtual Memory That is Larger Than Physical Memory
A Process’s Virtual-address Space

- stack
- heap
- data
- code

Max

0
Virtual Memory has Many Uses

In addition to separating logical from physical memory, **virtual memory can enable processes to share memory**

Provides following **benefits**:

- Can **share system libraries** among several processes
- Can share memory – **use for communication** between processes
- Can share pages **when create new process**, which results in faster process creation

**Copy-on-Write (COW)** allows both parent and child processes to **share** the same pages in memory initially

- 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
Shared Library Using Virtual Memory

- Stack
- Shared library
- Heap
- Data
- Code

Shared pages

- Stack
- Shared library
- Heap
- Data
- Code
Demand Paging – Lazy Swapper

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

- When page is needed – attempting to **reference** the page
  - If **invalid reference** -- exception
  - If **not-in-memory** -- bring in

- Now **called a pager** rather than a swapper, since does not operate on whole program, etc.

- Requires hardware support to know whether page table entry is **valid or invalid**
  - Valid / invalid refers to whether in mainstore or not
Swapping a Program to Contiguous Disk Space

- Initial concept of a **swapper** was a function that would swap **whole programs** in and out of memory.

- A **pager** swaps **individual pages**.
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
- Example of a page table snapshot:

```
<table>
<thead>
<tr>
<th>Frame #</th>
<th>valid-invalid bit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>:</td>
</tr>
<tr>
<td></td>
<td>:</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
```

- 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

The diagram illustrates a page table with some pages not in main memory. Each frame in the page table is labeled with the corresponding page from the logical memory. The valid-invalid bit indicates whether the page is in the main memory.

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

The physical memory contains pages A, B, C, D, E, F, G, and H, with some pages not present in the main memory.
What Is A Page Fault

- If there is a **reference to a page not in memory** (first reference or page has been swapped out) => exception (or trap) => **PAGE FAULT**
  - OS must decide:
    - **Invalid reference** – then different exception and abort.
    - **Page just not in memory**.

- Need to **handle the page fault** and go get the missing page to be able to continue running
  - Allocate free frame
  - Read page from disk into newly allocated frame
  - Update page table, set valid bit = 1
  - Restart instruction
    - Can do this because saved the state of the process when took original exception
    - But contains interesting challenges
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

Page Table

Free Frame

Physical Memory

Reference
Some instructions reference several addresses
- Could fault on any or all of the addresses – instruction partly done
- Some work may already have been done, data changed
- How is the system going to be able to reexecute instruction

Examples of where problems may occur:
- Complex move character instructions
- Autoincrement / autodecrement instructions

Solutions:
- Try to reference all addresses before executing instruction
- Use special registers to hold data changed, until instruction completed
- Use special registers to hold original state and data

And, of course, some code cannot fault at all
- e.g., the code that handles the page faults, I/O, or I/O errors
Also Implication On Performance

- Page Fault Rate $0 \leq p \leq 1.0$
  - if $p = 0$, no page faults
  - if $p = 1$, every reference is a fault

- Must add page fault handling to Effective Access Time (EAT)

  $$EAT = (1 - p) \times \text{memory access} + p \times$$
  $$(\text{page fault overhead} + \text{swap page out} + \text{swap page in} + \text{restart overhead})$$

- All this means that servicing page faults takes time, which affects the system's overall Effective Access Time (EAT)
Page Fault Overhead Details – Take Time

- Service the page fault exception
  - Exception itself
  - Save state of process that took the fault
  - Determine that exception was page fault
  - Determine if address reference valid
  - Find location of page on disk
  - Issue Read

- Read in the page
  - Wait in read queue
  - Seek / latency time
  - Move data from disk into memory

- Restart process that took the fault
  - Necessary, because have switched CPU to another process (to handle the fault)
  - Signal interrupt that I/O complete
  - Save state of current process (I/O process)
  - Determine what kind of interrupt (I/O complete – page fault handling complete)
  - Update page table, etc.
  - Restore state of original process that took the fault
  - Resume processing
Actual Impacts to Performance

- Page fault service time = 25ms => 30ms

- If 1 out of 1000 memory accesses causes a fault (i.e., .1% of the time), will slow down system by factor of 250.

- To reach 10% performance impact (only 10% slower) must only fault 1 of every 2,500,000 memory accesses.

- Why go to all this trouble
  - Can get many more users in mainstore at once
  - Can get better utilization of hardware
  - Can handle jobs bigger than mainstore
  - But it costs – performance & complexity
But It's Really Worse Than That (Sorry, I Lied)

- What if there is no free frame
  - Because of increased multiprogramming, memory is full

- Page replacement
  - Find some page in memory (victim) and write it out to disk (if changed), before OS can read in the page needed to handle the fault
  - Same page may be paged into memory and paged out again several times

- EAT = (1-P) * mem_access + P * (page_fault_overhead) + (select_and_write_out_victim_page)

- Page replacement algorithms (select which page will be paged out)
  - Will have significant performance impact
  - Want an algorithm that will result in a minimum number of page faults, not counting initial page reference
  - Actually, want to avoid paging out any pages until done using them
Need For Page Replacement

Note – memory is full – so when switch context to user 2 and try to reference page B, will need to replace a page
Additional Steps for Handling Page Fault

- Find the location on disk of the desired page

- Find a free frame
  - If there is a free frame, use it
  - If there is not a free frame, use a page replacement algorithm to select a victim frame
  - If victim frame has been changed, write it out to disk, which creates a free frame (and update that process’s page table)
  - Update the free frame table (allocate the page frame)

- Read the new page into the free frame, just allocated to the process
  - Update page table

- Restart the process
Page Replacement

- Frame: valid-invalid bit
- Page Table: 0 i, f v
- Physical Memory
- Swap out victim page
- Swap desired page in
- Change to invalid
- Reset page table for new page
Page Replacement Algorithms

- Decision of which page to replace is based on a Page Replacement Algorithm.

- Why are we interested???

- Because – we want lowest possible page-fault rate.
  - Affected by page reference string and number of frames.

- 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

Will evaluate paging algorithms with different numbers of available frames.
In general, more available frames = fewer faults required.
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)

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

9 page faults

- 4 frames

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

10 page faults

- FIFO Replacement – Belady’s Anomaly (1970, IBM Research)
  - Sometimes more frames may actually increase faulting rate
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 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

15 page faults

Will use this reference string to evaluate several algorithms
FIFO Illustrating Belady’s Anomaly

![Graph showing the number of page faults against the number of frames, illustrating Belady’s Anomaly. The graph has a line that shows a sudden increase in the number of page faults after a decrease, demonstrating the anomaly.](image-url)
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

  4 5

  6 page faults

- **BUT HOW DO YOU KNOW WHICH FRAME TO REPLACE ???**
  - **You can’t** – it is only useful as a theoretical goal – best possible case
  - Try to approach / approximate
  - Use as standard for measuring other algorithms
  - Can reconstruct based on logs of actual runs
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 7
0 0 0 0 4 0 0 0 0
1 1 3 3 3 1 1 1 1

9 page faults

FIFO = 15
Least Recently Used (LRU) Algorithm

- Victim is page that has not been used for longest period of time
  - Considered quite good

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

```
  1  5
  2
  3  5  4
  4  3
```

8 page faults

- Use this to try to approximate optimal
  - Question is, how to implement
LRU Page Replacement

reference string

\[
\begin{array}{ccccccccccccccc}
7 & 0 & 1 & 2 & 0 & 3 & 0 & 4 & 2 & 3 & 0 & 3 & 2 & 1 & 2 & 0 & 1 & 7 & 0 & 1 \\
7 & 7 & 7 & 2 & 2 & 4 & 4 & 4 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1
\end{array}
\]

page frames

\[
\begin{array}{ccccccccccccccc}
7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
\]

12 page faults

Optimal = 9
FIFO = 15
LRU Implementation Options

■ Associate *timestamp* with each page table entry
  ● Whenever page referenced, update timestamp
  ● Search timestamps to select victim
  ● Problem: *quite expensive*

■ Keep a *stack* or queue of page numbers in order of use
  ● When page referenced, move page number to top of stack
    ➢ Implement with doubly-linked list
  ● BUT – *is also expensive*
    ➢ Requires 6 pointers be changed every time a page is referenced - plus
      must traverse list to find page for each reference, and then move
    ➢ Advantage -- no search for replacement (just take bottom of stack)

■ Approximate using *Reference Bit* (quite common - *inexpensive*)
  ● Associate bit with each page
  ● When page referenced, bit set = 1
  ● Victim is first unreferenced page found
    ➢ Age or reset bits periodically
    ➢ Do not know order in which pages have been used, but probably OK
Use Of **Stack** to Record Most Recent Page References

reference string

\[4 \ 7 \ 0 \ 7 \ 1 \ 0 \ 1 \ 2 \ 1 \ 2 \ 7 \ 1 \ 2\]

- **stack before a**
  - 4
  - 7
  - 0
  - 1

- **stack after b**
  - 7
  - 2
  - 1
  - 0
Second Chance Algorithm

Good approximation of LRU if a little extra HW is available

- Need **reference bit** (and sometimes timestamp)

- Pages in **circular queue** (if no timestamp)
  - Examine page information:
    - If page unreferenced, then it is victim
    - If page referenced, reset reference bit and examine next page in queue
    - If get back around the queue to the page just reset, and have not found a victim yet, then it becomes the victim
  - **This is often the way LRU / Second chance is implemented, since**
    - Doesn't require special hardware support
    - Setting a bit is much quicker than getting and storing an entire timestamp
    - Testing / comparing bits is much quicker than working with timestamps
**Second Chance Page-Replacement Algorithm**

The Second Chance Page-Replacement Algorithm is a page replacement algorithm that aims to optimize the performance of the cache. Unlike other algorithms, such as the Least Recently Used (LRU) algorithm, Second Chance takes a more relaxed approach to deciding which page to evict from the cache. It keeps track of pages that have been referenced since the last time they were accessed, allowing it to give pages another chance before evicting them.

In the diagram, the algorithm is illustrated with a circular queue of pages. Each page has a reference bit that indicates whether it has been referenced since the last access. If a page is referenced, its reference bit is set to 1, allowing it to stay in the cache for another chance. If a page is not referenced, its reference bit remains 0, and it is considered for eviction.

The algorithm works as follows:

1. Whenever a page is referenced, its reference bit is set to 1.
2. If a page reaches the end of the queue and is not referenced, it is evicted from the cache.
3. The reference bit is not reset until the page is actually referenced again.

This approach helps to prevent unnecessary page faults and improves the overall performance of the system.
Additional technique to improve page fault service time

During page fault, it would help if only had to write out pages that had changed (writing out unchanged pages is waste of time)

Add and use modify (dirty) bit (in page table or elsewhere)
  - Set on when page has been changed, but not yet written
  - Only write out dirty pages
  - Also have one or more low priority background task(s) writing out changed pages and resetting dirty bit, as spare CPU resource is available (Page-out task)
## Combination Algorithms

- Can **combine use of reference bit and dirty bit** to improve performance even more

- **Priority for selecting a victim** then becomes:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not referenced, not dirty</td>
<td>0, 0</td>
<td>May not need again, and don’t have to write out</td>
</tr>
<tr>
<td>Not referenced, dirty</td>
<td>0, 1</td>
<td>May not need again, but do have to write out</td>
</tr>
<tr>
<td>Referenced, not dirty</td>
<td>1, 0</td>
<td>Likely will need again, but don’t have to write out</td>
</tr>
<tr>
<td>Referenced, dirty</td>
<td>1, 1</td>
<td>Likely will need again, and do have to write out</td>
</tr>
</tbody>
</table>
Many Other Algorithms Possible

- One category is **counting-based algorithms**
  - Keep a counter of the **number of references** that have been made to each page.
  - Look at reference counts to make decisions concerning victim

- **LFU Algorithm**
  - Victim is page that has been **used the least**
  - Problem if one page is used heavily at beginning and then never again

- **MFU Algorithm**
  - Victim is page **used the most**, since page just brought in will have been used the least

- **BUT -- neither approximates optimal very well**
Processes & Allocation of Frames

- Each process needs **minimum number of pages** in order to run.
  - Defined by architecture and instruction set

- Example: IBM 390 – 6 pages to handle SS MOVE instruction:
  - The instruction is 6 bytes, might span 2 pages.
  - The *from* address range could span 2 pages
  - The *to* address range could span 2 pages

- Most troublesome are architectures that allow **multiple levels of indirection**
  - Each level could reference a different page, and result in a fault

- Two major schemes for allocating frames to processes:
  - Fixed allocation
  - Priority allocation
Fixed Allocation Algorithms

- **Equal allocation**
  - E.g., if 100 frames and 5 processes, give each 20 pages.

- **Proportional allocation**
  - Allocate according to the size of process.

- **Priority Allocation**
  - Use proportional allocation based on priority rather than size
  - If high priority process faults, steal from lower priority process

- **Combinations** (especially size / priority)

- All affected by **multiprogramming level** (# processes in system)
  - All of this involved in “tuning” system performance
Global vs. Local Allocation

- **Global** page replacement
  - Process selects a replacement frame from the set of all frames
    - One process can steal a frame from another.
  - Problem – a process cannot control its own faulting rate
    - So, performance is not deterministic each time process runs

- **Local** page replacement
  - Each process selects from only its own set of allocated frames
    - Performance is more deterministic.
  - But this also means that other, less used pages (in other processes) are not available to it
    - So may reduce throughput

- Because global replacement usually results in better overall system throughput, it is more common

- Combinations possible (pools of processes / memory)
Thrashing

- If **too many processes are “active” at once**, a process may **not have “enough” pages** (i.e., minimum it needs to run **well**)
  - Page fault rate becomes quite high
  - CPU utilization becomes low
  - OS thinks it needs to increase level of multiprogramming
  - Adds another process to the system
  - Makes the situation worse

The result is:

Thrashing

Process is spending more time paging than in doing useful work
Thrashing
How to Handle Thrashing

- Control **maximum multiprogramming level**

- Use **local** rather than **global** page-replacement algorithm
  - A process can only steal pages from itself, not from another process

- Even with local replacement, still **must determine how many pages a process “needs”** to run efficiently
  - How?
  - Answer is based on concept of **LOCALITY**
Why Does Paging Work?

Review: why doesn't an entire process have to be in main memory at the same time in order to be able to run

- Processes execute **one instruction at a time**
- A process tends to **execute instructions near (in memory) to one another** – not just random instructions
  - Executes instructions in a procedure/method, then returns to main, then goes to another procedure/method, etc.
- **Locality** – set of pages used together
- **Locality model**
  - When executing, a process first runs in one locality and then in another
  - Localities may overlap
- If $\Sigma$ size of localities of all active processes $>$ total memory size, then **thrashing** occurs
Locality In A Memory-Reference Pattern
Working-Set Model

- \( \Delta \equiv \text{Working-Set Window} \equiv \text{a fixed number of page references} \)
  Example: 10,000 instruction

- \( WSS_i \) (Working Set Size of Process \( P_i \)) =
  total number of pages referenced in the most recent \( \Delta \)
  (varies in time)
  - if \( \Delta \) is too small, WS will not include entire locality
  - if \( \Delta \) is too large, WS will include parts of several localities
  - if \( \Delta = \infty \), WS will encompass entire program

- \( D = \sum WSS_i \equiv \text{total demand frames} \) for all processes on the system

- if \( D > \text{memory available on system} \) \( \Rightarrow \text{Thrashing} \)
  - So should suspend one of the processes
Working-set model

page reference table

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

\[
\begin{align*}
&\Delta \\
&\downarrow \\
&t_1
\end{align*}
\]

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

\[
\begin{align*}
&\Delta \\
&\downarrow \\
&t_2
\end{align*}
\]

\[
WS(t_2) = \{3,4\}
\]
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.
    - Reference bit and history bit
  - Whenever timer interrupts, copy reference bit to history bit and clear reference bit (but must do this for every page !!!)
  - If either bit is on, then page is in working set
  - Why is this not completely accurate?
  - Can improve with more history bits and more frequent interrupts, but costly

- **Working set model** quite good, and useful information for pre-paging, but it is clumsy to try to use it to control thrashing
  - Some systems use “invocation groups” for pre-paging

- A more direct approach is **Page Fault Frequency**
Page-Fault Frequency Scheme

Problem – how to control thrashing, but maximize CPU utilization and throughput

- Establish “acceptable” page-fault rate range
- If actual rate too low, process loses frame
- If actual rate too high, process gains frame
- Suspend a process if there are no free frames
Working Sets and Page Fault Rates

![Graph showing working sets and page fault rates over time.](image)
Other Considerations

- **Global vs. local page replacement**
  - With **global**, process can't control its own faulting rate
    - Different characteristics each time it runs
  - With **local**, can get inefficient use of resources
  - **Combinations** possible (pools of pages for similar type processes)

- **Prepaging** – bring all pages process will need
  - **Difficult to predict** – no good algorithms (without history or a priori data)
  - Useful if select “right” set of pages, and then pre-bring is less costly than faulting in one at a time (Activation group, Invocation group)
  - Also useful for data resources (files, etc.)

- **Page size selection** – system page size will affect:
  - Page table size
  - Internal fragmentation
  - I/O overhead
  - Locality
  - **General architectural trend** -- page size is increasing, since faulting is becoming more costly, and have limited space for page tables and TLB
  - Sometimes multiple page sizes supported on a single system image
Other Considerations (Cont.)

- **I/O Interlock and addressing (Locked / Pinned pages)**
  - Sometimes do not want a frame to be paged out no matter what
    - Code required to do the I/O to handle faults / errors
    - Buffers
    - Partially updated pages
    - Critical kernel code
    - Performance critical data / tables
  - Solution – lock or “pin” pages in memory
    - Can't be paged out until unlocked / unpinned

- **Inverted page table**
  - To reduce size of page table, use single page table, not one for each process
  - Page table ordered by virtual addresses of pages in memory
    - Is relatively small – use hash table to find entries
  - Complex, but minimizes resources

- **Demand segmentation**
  - Approximate demand paging if paging support hardware is not available – much overhead -- expensive
Reason Why Frames Used For I/O Must Be In Memory

buffer

disk drive
### Additional Options

- **Ideally, the working set of each process is stored in the TLB.**
  - Otherwise there will be a high degree of page faults.
    - **TLB Reach** = (TLB Size) \( \times \) (Page Size)
  - If it's not, what can be done?

- **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.
  - Also allows large blocks of code or modules that must stay in memory, to only take up one entry in the TLB
  - But adds complexity

- **Increase size of TLB**
  - But note – **TLB is HW dependent, so expensive to change**
Other Impacts on Paging

- **Process creation can be speeded up** by several techniques, made possible with virtual memory:
  - Copy-on-Write
  - Memory-Mapped Files

- These, however, also **can impact paging**
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**, therefore, allows *more efficient process creation*
  - No pages are copied when process is created
  - Only modified pages are copied, and only later, after process is running

- Free pages are allocated from a *pool* of zeroed-out pages.

- **BUT** – increases *complexity* of page replacement decisions
  - Now must consider whether *either* process has referenced or changed a page
Before Process 1 Modifies Page C
After Process 1 Modifies Page C
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 (and speeds up) file access by treating file I/O through memory rather than through `read()` `write()` system calls.

- Also allows several processes to map the same file allowing the pages in memory to be shared. (major use of memory mapped files)

- Can significantly increase the challenges for the OS for both paging and synchronization
Memory Mapped Files

- Process A virtual memory
- Process B virtual memory
- Physical memory
- Disk file
Memory-Mapped Shared Memory in Windows

process$_1$

shared memory

memory-mapped file

shared memory

process$_2$

shared memory
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

$B_R$

32 KB

$C_L$

32 KB

$C_R$
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
- 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
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 $s \times \alpha$ save pages faults $> \text{or} < \text{than the cost of prepaging}$
    - $s \times (1-\alpha)$ 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
  - 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
Why Should You Care ???

- You can impact paging and performance if you consider the locality / working set of your code as it runs
  - Know where code for procedures, functions, methods is placed
  - Know how data structures / objects layout in memory

Example: int a[1024][1024]; => one row per page / frame

```c
for(int i = 0; i < 1024; ++i) {
    for(int j = 0; j < 1024; ++j) {
        a[i][j] = 0;
    }
} // 1024 page faults
```

```c
for(int j = 0; j < 1024; ++j) {
    for(int i = 0; i < 1024; ++i) {
        a[i][j] = 0;
    }
} // 1024 * 1024 page faults
```
End of Chapter 9