Memory Management

Instructor: Dr. Tongping Liu

Outline

- Virtual memory
- Page-based memory management
  - Page table and address translation
  - Multi-level page table
  - Translation lookaside buffer (TLB)
  - Demand paging
  - Thrashing and working set
  - Page replacement
  - Kernel memory management
  - User-space memory management

Memory

- The ideal memory is
  - Very large
  - Very fast
  - Non-volatile (doesn’t go away when power is turned off)

- The real memory is
  - Not very large
  - Not very fast
  - Affordable (cost)!

- Memory management goal: make the reality similar to the ideal world as much as possible 😊

Virtual Memory

- Basic idea: allow OS to allocate more memory than the real
- Program uses virtual addresses
  - Addresses local to the process
  - Limited by # of bits in address (32/64)
  - 32 bits: 4GB
  - 64 bits (actually 48 bits): 256TB
  - Virtual memory >> physical memory
Motivations for Virtual Memory

- Efficient Use of Limited Memory
  - Keep only active address space in the memory
  - Some non-active parts can be stored on the disk
- Simplify Memory Management
  - Each process “gets” the same full, linear address space
- Isolates Executions of Multiple Processes
  - One process can’t interfere with another’s memory
    - They operate in different address spaces
  - User process cannot access privileged information
    - Different sections of address spaces have different permissions

Virtual and Physical Addresses

- Virtual address space
  - Determined by the instruction width
  - Same for all processes
- Physical memory indexed by physical addresses
  - Limited by bus size (# of bits)
  - Amount of available memory

Outline

- Virtual memory
- Page-based memory management
  - Page table and address translation
- Multi-level page table
- Translation lookaside buffer (TLB)
- Demand paging
- Thrashing and working set
- Page replacement
- Kernel memory management
- User-space memory management

Page-Based Memory Management

- Virtual address
  - Divided into pages
- Physical memory
  - Divided into frames
- Page vs. Frame
  - Same size address block
  - Unit of mapping/allocation
- A page is mapped to a frame
  - All addresses in the same virtual page are in the same physical frame
  - Offset in a page
Page Table

- Each process has one page table
  - Map page number to physical frame number
  - Number of PTEs in page table
  - Number of total pages in virtual space
  - Not just the pages in use

- Page table is checked for every address translation
- Where to store page table?

- Not all pages need to be mapped to frames at the same time
- Not all physical frames need to be used

Translate Virtual to Physical Address

- Split virtual address (from CPU) into two pieces
  - Page number ($p$)
  - Page offset ($d$)

- Page number
  - Index into an entry of the page table that holds the corresponding physical frame number

- Page offset
  - Position inside a page

- Page size = $2^n$ bytes: determined by offset size

Logic Address

- Suppose logical address space is $2^m$ and page size is $2^n$, so the number of pages is $2^m/2^n$, which is $2^{m-n}$

- Logical Address ($m$ bits) is divided into:
  - Page number ($p$) – used as an index into a page table which contains frame number of physical memory
  - Page offset ($d$) – combined with base address to define the physical memory address that is sent to the memory unit

- Page #: 4 bits
- Offset: 12 bits

An Example of Virtual/Physical Addresses

- Example:
  - 64 KB virtual memory
  - 32 KB physical memory
  - 4 KB page/frame size → 12 bits as offset ($d$)

- Virtual address: 16 bits

- How many virtual pages?

- Physical address: 15 bits

- How many physical frames?
In-Class Exercise

For an embedded small computer supporting up to 1k bytes physical memory, where its virtual address is 12 bits. Suppose that the size of page/frame is 128 bytes. Use a figure to illustrate the physical and virtual address that shows the number of bits for offset and frame/page number for one level page table.

- What is the number of virtual pages for each process?
- How many physical frames in total?
- How many entries in page table for each process?

Address Translation

Example:
- 64 KB virtual memory
- 32 KB physical memory
- 4 KB page/frame size → 12 bits as offset (d)

How many virtual pages?
How many physical frames?
Virtual address 0x44, offset 0x44

1. Compute the page number
   Page number is #0

2. Check the page table, get physical frame number
   Frame number is #2

3. Compute the starting address of physical frame
   Start Address = 2 * 0x80 = 0x100

4. Compute the physical address
   Physical Address = 0x100 + 0x44 = 0x144

Virtual address 0x224, offset 0x24

1. Compute the page number
   Page number is #4

2. Check the page table, get physical frame number
   Frame number is #3

3. Compute the starting address of physical frame
   Start Address = 3 * 0x80 = 0x180

4. Compute the physical address
   Physical Address = 0x180 + 0x24 = 0x1A4
Limitations of One-level Page Table

- Page table should contain all pages of virtual space
  - Otherwise, some addresses can't be accessed.
- All entries should be physically continuous
  - Proof by contradiction. If the index is not continuous, we will need another mechanism to tell us where has a hole or not

Page/Frame Size

- Determine the **number of bits in offset**
- Smaller page size
  - + Less internal fragmentation
  - - Larger page table: require more than one frame, hard to allocate!
- Larger page size
  - + Smaller page table and less overhead to track them
  - + More efficient to transfer to/from disk
  - - More internal fragmentation, wasting memory

Size of Page Table

- If a computer has 22 bits of logical address (4 MB), and supports the physical memory up to 1 MB. The page/frame size is 1KB.
  - Virtual addr: P#: 12 bits Offset: 10 bits
  - Physical addr: F#: 10 bits Offset: 10 bits
  - Number of pages: 12 bits → 4K pages
  - Size of page table:
    - Each page entry must have at least 10 bits → 2 bytes
    - 4K * 2 bytes = 8KB → requires 8 consecutive frames

- Modern Machine
  - 32 bits virtual address
  - Size of one page/frame is 4KB (12 bits)
  - Page table size
    - # of virtual pages: 32 – 12 = 20 bits → $2^{20}$ PTEs
    - Page table size = PTE size * $2^{20}$ = 4 MB per process → $2^{20}$ frames
  - If there are 128 processes, and physical memory is 1 GB
    - Page tables occupy 128 * 4MB = 512 MB
    - 50% of memory will be used by page tables

*How can we get smaller page table?!*
Outline

- Virtual memory
- Page-based memory management
  - Page table and address translation
- Multi-level page table
- Translation lookaside buffer (TLB)
- Demand paging
- Thrashing and working set
- Page replacement
- Kernel memory management
- User-space memory management

Two-Level Page Tables

- Solution: multi-level page tables
- Virtual address: three parts
  - Level-one page number (10 bits)
  - Level-two page number (10 bits)
  - Offset (12 bits)
- PTE in 1st level page table contains the address of physical frame for 2nd level page table
- PTE in 2nd level page table has the actual physical frame number

Example: 2-level Address Translation

- We don’t have to allocate all entries initially
- They don’t have to be continuous
Memory Requirement of Page Tables

Only the 1st level page table and the required 2nd level page tables need to be in memory

32bit machine, page size 4k, each entry 4 bytes, using two-level page table (with 10bit-10bit-12bit design), what is the size for the full page table

- Level-0: 1024 entries*4bytes
- Level-1: 1024*1024 entries=1M entries * 4bytes
- Total: 4M+4K

Example: if a process accesses 32 MB virtual memory on such a machine, what is the minimum memory required for the page table

- 4KB / page → process has 8K (8*2^10) virtual pages
- One 2nd level page table maps 2^10 pages;
- Number (minimum) of 2nd level page table needed: 8*2^10/2^10 = 8
- Total (minimum) memory for page table: 1st level page table + 8; in total of 9 page tables → 9 X 4KB = 36 KB

Other Questions

- If we have two pages (0x00000000 and 0x00201000), what is the size of page table?
- 32bit machine, page size 4k, each entry 4 bytes, two level page table (two pages:0x00000000, and 0xFFFFF000)
Linux’s 3 level page table

Linear Address converted to physical address using 3 levels

- Index into Page Dir.
- Index into Page Middle Dir.
- Index into Page Table
- Page Offset

What are the benefits to use 3-level page table?
What are the shortcomings?

Designing of 2-level page table

1. Determine number of bits of page offset
   (12Bit→4KB)
2. Determine the size of each page table entry
3. Determine the number of bits for 2nd level page table
   **One principle:** page table → fit into one frame
4. Determine the number of bits for 1st level page table

Page Table Entry (PTE)

- Each entry in the page table contains
  - Frame number: bits depends on # of frames in physical memory
  - Valid bit: set if this logical page number has a corresponding physical frame in memory
    - If not valid, remainder of this PTE is irrelevant
  - Referenced bit: set if data on the page has been accessed
  - Dirty (modified) bit: set if data on the page has been modified
  - Protection information: read-only/read-write
- Size of each PTE: at least frame number plus 4/5 bits

Designing of 2-level Page Table

- Suppose that a system has a 24-bit logical address space and is byte-addressable. **Physical address has 20 bits,** and the size of a page/frame is 1K bytes. How to design a two-level page table?
  1. The size of page table entry will be 2 bytes (larger than 15 bits)
  2. One page can hold 512 entries (1K/2=512)
  3. Thus, we need 9 bits for the 2nd level page table
Designing of 2-level Page Table

Suppose that a system has a 24-bit logical address space and is byte-addressable. Physical address has 20 bits, and the size of a page/frame is 1K bytes. How to design a two-level page table?

Second level 10 bits, can’t be fitted into one page. Then we may need to have multiple pages (2 pages) that are continuous, which increases the complexity of OS design.

Designing of 2-level Page Table (2)

Suppose that a system has a 18-bit logical address space and is byte-addressable. The amount of physical memory is 64KB (i.e., the physical address has 16 bits) and the size of a page/frame is 128 bytes. How to design a two-level page table?

1. The size of page table entry will be 2 bytes (larger than 11 bits)
2. One page can hold 64 entries ($128/2=64$)
3. Thus, we need 6 bits for the 2nd level page table

How Many Levels are Needed?

New architecture with 64-bits address?

- Suppose 4KB page/frame size (12-bit offset)
- One-level page table, with $2^{12} / 2 = 2^{10}$ entries!!!
- Two-level page table, then inner page table could be one page, containing $1K$ entries (10-bit) (assuming PTE size is 4 bytes), then needs to contain $2^{10}$ entries!!!
- In total, we will have 6-level paging

Problems with multiple-level page table

- One additional memory access for each level added
- Multiple levels are possible; but prohibitive after a few levels

Are there any other alternatives to manage the page table, in order to reduce access time?

Outline

- Virtual memory
- Page-based memory management
  - Page table and address translation
- Multi-level page table
- Translation lookaside buffer (TLB)
- Demand paging
- Thrashing and working set
- Page replacement
- Kernel memory management
- User-space memory management
Translation Look-ahead Buffers (TLB)

Translation Look-ahead buffers (TLB)
- A special fast-lookup hardware cache: associative memory
- Access by content → parallel search: if in TLB, get frame #
- Otherwise, access the page table in memory, and get frame # and put it into TLB
  (if TLB is full, replace an old entry.)

Access by content: parallel search
- If in TLB, get frame #
- Otherwise, access the page table in memory, and get frame # and put it into TLB
  (if TLB is full, replace an old entry.)

Translation Lookaside Buffer (TLB)

Small Hardware: fast
- Store recent mapping of page → frame (64 ~ 1024 entries)
- If desired logical page number is found, get frame number from TLB
- If not,
  - Get frame number from page table in memory
  - Use standard cache technique
  - Replace an entry in the TLB with the logical & physical page number from this reference
  - May contain complete page table entries for processes with small number of pages

Paging Hardware With TLB

Address Translation with TLB

What happens when CPU performs a context switch?
Integrating VM and Cache

- Modern caches are “physically addressed”
  - Allows multiple processes to have data in the cache simultaneously. Otherwise, every context switch will require to flush the cache.
  - Cache doesn’t need to be associated with protection issues.
  - Access rights checked as part of address translation.
- Perform address translation before cache lookup
  - Could involve memory access (to get PTE)
  - So page table entries can also be cached.

Integrating TLB, Cache with Memory

“Translation Lookaside Buffer” (TLB)

Basic overflow

1. CPU will generate a virtual address.
2. Check the TLB at first to find the mapping of this page. If the corresponding entry exists, then we could get the PA address directly. Otherwise, we will go through the slow address translation, and then save the mapping into TLB (typically after the translation).
3. After getting a PA address, check whether the corresponding cache line is in the cache or not. If yes, then we will have a cache hit. The value of the corresponding memory unit will be returned to CPU.
4. Otherwise, it is a cache miss. The cache line will be fetched into the cache at first. Then the value will be sent back to back to the CPU.

Memory Accesses Time

- Assuming:
  - TLB lookup time = a
  - Memory access time = m
- Hit ratio (h) is percentage of time that a logical page number is found in the TLB
  - More TLB entries usually means higher h
- Effective Access Time (EAT) is calculated (don’t include cache effect)
  - EAT = (m + a)h + (m + m + a)(1-h) = a + (2-h)m
- Interpretation
  - Reference always requires TLB lookup, 1 memory access. TLB misses also require an additional memory reference.
Effective Access Time: An Example

- TLB Lookup = 20 nanosecond
- Memory cycle time is 100 nanosecond
- Hit ratio = 80%
- Effective Access Time (EAT) would be
  \[ EAT = (20 + 100) \times 0.8 + (20 + 100 + 100)(1 - 0.8) \]
  \[ = 140 \text{ nanosecond} \]
- 40% slow down
- What is the EAT with 98% hit rate?

Outline

- Virtual memory
  - Page-based memory management
    - Page table and address translation
  - Multi-level page table
  - Translation lookaside buffer (TLB)
- Demand paging
  - Thrashing and working set
  - Page replacement
  - Kernel memory management
  - User-space memory management

Page Table Entry (PTE)

- Each entry in the page table contains
  - Frame number: bits depends on # of frames in physical memory
  - Valid bit: set if this logical page number has a corresponding physical frame in memory
  - Referenced bit: set if data on the page has been accessed
  - Dirty (modified) bit: set if data on the page has been modified
  - Protection information: read-only/read-write
- Size of each PTE: at least frame number plus 4/5 bits

Valid (v) or Invalid (i) Bit
Paging and Demand Paging

Paging:
- **Paging** is a memory management scheme by which a computer stores and retrieves data from secondary storage for use in main memory with the same-size blocks called pages.
- **Paging** lets programs exceed the size of available physical memory.

Demand Paging:
- Pages are loaded only when they are referenced.

The valid bit allows the implementation of demand paging.

Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated,
- **v** ⇒ in-memory,
- **i** ⇒ not-in-memory
- Initially bit is set to **i** on all entries
- During address translation, if valid–invalid bit in page table entry is **i** ⇒ page fault (trap)

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 ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory

Page Fault

1. Reference to a page, if Invalid reference ⇒ abort
2. If not in memory, page fault occurs (trap to OS)
3. OS allocates an empty frame
4. Swap page into frame
5. Reset page tables, set validation bit = **v**
6. Restart the instruction that caused the page fault
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)
  \[ EAT = (1 - p) \times \text{memory_access} + p \times \text{page_fault_time} \]
  - page_fault_time depends on several factors
  - Save user reg and proc state, check page ref, read from the disk there might be a queue, (CPU can be given to another proc), get interrupt, save other user reg and proc state, correct the page table, put this process into ready queue..... Due to queues, the page_fault_time is a random variable

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- \[ EAT = (1 - p) \times 200 + p \times (8 \text{ milliseconds}) \]
  \[ = (1 - p) \times 200 + p \times 8,000,000 \]
  \[ = 200 + p \times 7,999,800 \]
  - If one out of 1,000 access causes a page fault, then \[ EAT = 8199.8 \text{ nanoseconds.} \]
    - This is a slowdown by a factor of 40!!
  - If we want just 10% performance degradation, then $p$ should be $220 > (1 - p) \times 200 + p \times (8 \text{ milliseconds})$
    \[ p < 0.0000025 , \text{i.e., 1 page fault out of 400,000 accesses} \]

Outline

- Virtual memory
- Page-based memory management
  - Page table and address translation
- Multi-level page table
- Translation looksaside buffer (TLB)
- Demand paging
  - Thrashing and working set
- Page replacement
- Kernel memory management
- User-space memory management

Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high.
  - E.g. a process needs 6 pages, but only have 5 frames. Thus it will evict a page from existing 5 pages. \rightarrow frequent faults
  - This leads to:
    - low CPU utilization
    - OS increase the degree of multiprogramming
    - another process added to the system, worse case
Locality and Thrashing

To prevent thrashing we should give enough frames to each process.
But how much is "enough"?
Locality model:
- locality: a set of pages that are actively used
  - Process migrates from one locality to another
  - Localities may overlap
- When $\sum$ size of locality $> \text{total memory size}$, thrashing occurs...

Increase locality in your programs!

Why locality model

- Suppose we allocate enough frames for a process to accommodate its locality.
  - It will fault for the pages in its locality until all these pages are in memory. Then, it will not fault again until it changes its localities.
- Otherwise, if we do not allocate enough frames, the process will thrash, since it cannot keep in memory all the pages that it is actively using.
- The working-set model is based on the assumption of locality.

Working-Set Model

- $\Delta$ = working-set window = 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 $\Delta$
  - if $\Delta$ too small, will not encompass entire locality
  - if $\Delta$ too large, will encompass localities
  - if $\Delta = \infty$, will encompass entire program
- $D = \sum WSS_i$ = total demand frames
  - if $D > \text{(available frames)}$, then thrashing
    - Thus, if $D > m$, then suspend one of the processes

Working-Set Definition

- Informal Definition: the collection of pages that a process is working with, and which must thus be resident if the process is to avoid thrashing.
- The idea is to use the recent needs of a process to predict its future needs.
  - Choose $\Delta$, the working set parameter. At any given time, all pages referenced by a process in its last $\Delta$ seconds of execution are considered to comprise its working set.
  - Pages outside the working set may be discarded at any time.
Outline

- Virtual memory
- Page-based memory management
  - Page table and address translation
- Multi-level page table
- Translation lookaside buffer (TLB)
- Demand paging
- Thrashing and working set
- Page replacement
- Kernel memory management
- User-space memory management

Page Replacement

- No free frames in memory, a page needs to be replaced.
- Pages that are replaced might be needed again later.
- We need algorithms to minimize the number of page faults.
- Include other improvements, e.g., use modify (dirty) bit to reduce overhead of page transfers – only modified pages are written to disk

Basic Page Replacement

- Find the desired page on disk
- If there is a free frame, use it
- If there is no free frame, use a page replacement algorithm
  1. Select a victim frame, swap it out (use dirty bit to swap out only modified frames)
  2. Bring the desired page into the (newly) free frame;
  3. Update the page and frame tables
- Resume the process

Page Replacement Algorithms

- How to select the victim frame?
  - You can select any frame, the page replacement will work; but the performance???
  - Gives the lowest page-fault rate
- Evaluate an algorithm by running it on a particular string of memory references (reference string) and compute the number of page faults on that string In all our examples, we will have 3 frames and the following reference string

```
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
```
First-In-First-Out (FIFO) Algorithm

- Maintain an FIFO buffer
  - The page used before may not be needed
  - An array used early, might be used again and again
- Easy to implement
- Belady’s Anomaly: more frames ⇒ more page faults

<table>
<thead>
<tr>
<th>Reference string</th>
<th>7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page frames</td>
<td><img src="image" alt="Page frames" /></td>
</tr>
</tbody>
</table>

FIFO Illustrating Belady’s Anomaly

Reference string (12 accesses)
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Optimal Algorithm

- Replace a page that will not be used for longest time

<table>
<thead>
<tr>
<th>Reference string</th>
<th>7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page frames</td>
<td><img src="image" alt="Page frames" /></td>
</tr>
</tbody>
</table>

Least Recently Used (LRU) Algorithm

- Use recent past as an approximation of the future
- Select the page that is not used for a long time...
  - OPT if you look at from backward
  - NO Belady’s Anomaly: so more frames ⇒ less page faults

<table>
<thead>
<tr>
<th>Reference string</th>
<th>7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page frames</td>
<td><img src="image" alt="Page frames" /></td>
</tr>
</tbody>
</table>
Given the reference string of page accesses: 1 2 3 4 2 3 4 1 2 1 3 1 4 and a system with three page frames, what is the final configuration of the three frames after the true LRU algorithm is applied?

Problem of LRU:
How to implement it efficiently?
Full LRU needs to sort all references

**Implementing LRU Algorithm**

- **Counter (logical clock) implementation**
  - Increase the counter every time a page is referenced
  - Save it into time-of-use field
  - Find one with the smallest time-of-use value
  - Problems: Counter overflow and linear search overhead
- **Stack implementation**
  - Keep a stack of page numbers in a double link form:
    - Page referenced: move it to the top
    - requires 6 pointer ops to be changed
    - No search for replacement
  - Least recently used one is at the bottom
- **Second Chance Algorithm (LRU Approximation)**
  - Uses one reference bit and one pointer
  - The reference bit will be set to 1 if a page is referenced
  - Pointer pointing to the one that is be checked next
  - If ref bit is 0,
    - replace that page
    - Move the pointer to the next one
  - If ref bit is 1, "give a second chance”
    - set ref bit to 0
    - Leave page in memory
    - go to next one
- **Circular queue implementation:**
  - A pointer indicates which page will be replaced next
  - When a frame is needed, it advances its pointer until find one with a 0 reference bit.

What if all bits are 1? … All pages will get second chance … Degenerates FIFO
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
  - High priority processes can take all frames from low priority ones (cause thrashing)
  - A process cannot control its page fault rate

- **Local replacement** – each process selects from only its own set of allocated frames
  - Consistent performance
  - Lower utilization of memory and less throughput

Summary: Page Replacement Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO (First-In, First-Out)</td>
<td>Might throw out useful pages</td>
</tr>
<tr>
<td>Second chance</td>
<td>Big improvement over FIFO</td>
</tr>
<tr>
<td>LRU (Least Recently Used)</td>
<td>Excellent, but hard to implement exactly</td>
</tr>
<tr>
<td>OPT (Optimal)</td>
<td>Not implementable, but useful as a benchmark</td>
</tr>
</tbody>
</table>

Outline

- Virtual memory
- Page-based memory management
  - Page table and address translation
- Multi-level page table
- Translation lookaside buffer (TLB)
- Demand paging
- Thrashing and working set
- Page replacement
- Kernel memory management
- User-space memory management
Kernel Memory Allocation

- E.g., Linux PCB (struct task_struct)
  - > 1.7 Kbytes each
  - Created on every fork and every thread create (e.g., clone())
  - Deleted on every exit

- Kernel memory allocators
  - Buddy system
  - Slab allocation

Buddy System (Dividing)

Two continuous blocks with the same size, the first one will start as $2^n$
---

**Buddy Allocation**

- Example: Need to allocate 65 contiguous page frames.
  - Look in list of free 128-page-frame blocks.
  - If free block exists, allocate it, else look in next highest order list (here, 256-page-frame blocks).
  - If first free block is in 256-page-frame list, allocate a 128-page-frame block and put remaining 128-page-frame block in lower order list.
  - If first free block is in 512-page-frame list, allocate a 128-page-frame block and split remaining 384 page frames into 2 blocks of 256 and 128 page frames. These blocks are allocated to the corresponding free lists.

- Question: What is the worst-case internal fragmentation?

---

**Buddy De-Allocation**

- When blocks of page frames are released the kernel tries to merge pairs of “buddy” blocks of size $b$ into blocks of size $2b$.
- Two blocks are buddies if:
  - They have equal size $b$.
  - They are located at contiguous physical addresses.
  - The address of the first page frame in the first block is aligned on a multiple of $2b^{2^{12}}$ (starting at 2b page)
- The process repeats by attempting to merge buddies of size $2b$, $4b$, $8b$ etc…

---

**Slab Allocator**

- Performs the following functions
  - Allocate memory
  - Initialize objects/structures
  - Use objects/structures
  - Deconstruct objects/structures
  - Free memory
- `/proc/slabinfo` – gives full information about memory usage on the slab level. (see also `/usr/bin/slabtop`)

---

---

---

---

---

---

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 (process descriptions, file objects, semaphores)
  - 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**, next object is allocated from empty slab. If no empty slabs, new slab allocated

Benefits include:
- No fragmentation
- Memory request is satisfied quickly

Outline

- **Virtual memory**
- Page-based memory management
  - Page table and address translation
- Multi-level page table
- Translation lookaside buffer (TLB)
- Demand paging
- Thrashing and working set
- Page replacement
- Kernel memory management
- User-space memory management

Memory allocation (using mmap/brk)

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    int * ptr = malloc(4);  
    *ptr = 1;
    free(ptr);
}
```

Currently, no heap space at all because we didn’t use any heap

Memory allocation

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    int * ptr = malloc(4);  
    *ptr = 1;
    free(ptr);
}
```

Now, the heap is allocated from the kernel, which means the virtual address from 0x0804b000 to 0x0806c000 (total 33 pages) are usable. It's actually 0x804b008.
# Memory Mapping (mmap or brk)

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int * ptr = malloc(4);
    *ptr = 1;
    free(ptr);
}
```

## What We Learn Here

- Typically, the user will ask one big block of memory and setup its page table.
- Then this block of memory will be managed by user space memory manager.
  - How to manage the memory inside user space?
Memory Allocator for Multithreaded Programs (Hoard)

- Shows the memory in-use
- Uses free-list to manage available/freed objects

Default Linux Allocator

- Designed to perform well
  - Bump pointers + freelists
- Not designed for security purposes
- Prepends metadata before actual objects
- Provides no randomization
  - Result: easy to determine next object to be allocated
- Poor handling of double/invalid frees

Outline

- Virtual memory
- Page-based memory management
  - Page table and address translation
- Multi-level page table
- Translation lookaside buffer (TLB)
- Demand paging
- Thrashing and working set
- Page replacement
- Kernel memory management
- User-space memory management