CS 5523 Operating Systems: Memory Management

Instructor: Dr. Tongping Liu

Thank Dr. Dakai Zhu, Dr. Palden Lama, and Dr. Tim Richards (UMASS) for providing their slides.

Outline
- Simple memory management: swap etc.
- Virtual memory and paging
  - Page table and address translation
  - Translation lookaside buffer (TLB)
- Multi-level page table
- Page replacement algorithms and modeling
- Working set of processes

Memory Hierarchy
- CPU can directly access main memory and registers only
- But programs and data must be brought (from disk) into memory
- Memory accesses can be the bottleneck
  - Cache between memory and CPU registers
- Memory Hierarchy
  - Cache: small, fast, expensive; SRAM
  - Main memory: medium-speed, not that expensive; DRAM
  - Disk: many gigabytes, slow, cheap, non-volatile storage

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)!
  - Pick any two...
- Memory management goal: make the real world look as much like the ideal world as possible

CPU

Main Memory

Cache

Disk

I/O Devices
Limitations without virtual memory

Protection of memory: using base and limit registers

Many Questions:
1. How to generate the addresses for a process?
2. How to assign physical memory?

Swapping

Consider a multi-programming environment:
- Each program must be in the memory to be executed
- Processes come into memory and
- Leave memory when execution is completed

Swapping would be needed to free up memory for additional processes.
Outline

- Simple memory management: swap etc.
- Virtual memory and paging
  - Page table and address translation
  - Translation lookaside buffer (TLB)
- Multi-level page table
- Kernel memory management
- Working set of processes

Virtual Memory

- Basic idea: allow OS to allocate more memory than the real
- Program uses virtual addresses
  - Addresses local to the process
  - Can be any size limited by # of bits in address (32/64)
    - 32 bits: 4G
    - 64 bits: 2^64
- Virtual memory >> physical memory

Motivations for Virtual Memory

- Use physical DRAM as cache for the disk
  - Virtual pages of processes can exceed physical memory size
- Simplify memory management
  - Multiple processes resident in main memory
    - Each with its own address space
  - Only “active” code and data is actually in memory
- Provide protection
  - One process can’t interfere with another
    - Because they operate in different address spaces
  - User process cannot access privileged information
    - Different sections of address spaces have different permissions

Virtual Memory for Multiprogramming

- Virtual memory (VM) is helpful in multiprogramming
  - Multiple processes in memory concurrently
  - Each process occupies small portion of memory
  - CPU schedules process B while process A waits for its long I/O operations (e.g., retrieve data from disks)
- Physical Memory de/allocation
  - Keep recently used content in physical memory
  - Move less recently used stuff to disk
  - Movement to/from disk handled by the OS

How to get physical address from the virtual one?!
Virtual and Physical Addresses

- Virtual address space
  - Determined by instruction width
  - Same for all processes
- Physical memory indexed by physical addresses
  - Limited by bus size (no of bits)
  - Amount of available memory

Paging: a memory-management scheme that permits address space of process to be non-continuous.

Paging and Page Systems

- 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, why?
- Page table is checked for every address translation
  - Where to store page table?
- Not all pages need map to frames at the same time
- Not all physical frame need 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, with addresses of physical pages
- Page offset: Position inside a page
- Page size = $2^d$ bytes: determined by offset size
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)

Page #: 4 bits
Offset: 12 bits
Virtual address: 16 bits

Frame #: 3 bits
Offset: 12 bits
physical address: 15 bits

Address Translation

Virtual address: 16 bits

How many virtual pages?

physical address: 15 bits

How many physical frames?

Virtual address: 16 bits

Physical frame: 12 bits

Project 1 is posted, due on 27th

A tiny computing system with 1K bytes physical memory, and the virtual address has 12 bits (4096 bytes). Suppose that the size of virtual page/frame is 128 bytes (i.e., with 7 bits in the address as offset).

- What is the number of virtual pages for each process? 32 pages
- How many physical frames in total? 8 frames
- How many entries in the page table for each process? 32

More on Page Table

Different processes have different page tables
- CR3 points to the page table
- Change CR3 registers when context switches

Page table resides in main (physical) memory
- Continuous memory segment

Why?
Address Translation Architecture

How big the page table is?

Examples for Address Translation

1K bytes physical memory, and the virtual address has 12 bits, and the size of a page is 128 bytes

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0044</td>
<td></td>
</tr>
<tr>
<td>0x0224</td>
<td></td>
</tr>
<tr>
<td>0x0136</td>
<td></td>
</tr>
</tbody>
</table>

1. Size of a page (128 B)
2. Page index and offset (0, 0x44)
3. Frame index (2)
4. Starting address of the frame (2 * 0x80)
5. Physical address (2 * (0x80 + 0x44)) = 0x144

Examples for Address Translation

1K bytes physical memory, and the virtual address has 12 bits, and the size of a page is 128 bytes

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0044</td>
<td></td>
</tr>
<tr>
<td>0x0224</td>
<td></td>
</tr>
<tr>
<td>0x0136</td>
<td></td>
</tr>
</tbody>
</table>

1. Size of a page (128 B)
2. Page index and offset (0, 0x44)
3. Frame index (2)
4. Starting address of the frame (2 * 0x80)
5. Physical address (2 * (0x80 + 0x44)) = 0x144
Page Table Size for 32bit System

Modern Systems/Applications
- 32 bits virtual address
- System with 1GB physical memory → 30 bits physical address
- Suppose the size of one page/frame is 4KB (12 bits)

Page table size
- # of virtual pages: \(32 - 12 = 20\) bits
- Page table size = PTE size \(\times 2^{20}\) = 4 MB per process \(\times 2^{12}\) frames

If there are 128 processes
- Page tables occupy 128 \(\times 4MB = 512\) MB
- 50% of memory will be used by page tables?

How can we get smaller page table??

Outline

- Simple memory management: swap etc.
- Virtual memory and paging
  - Page table and address translation
  - Translation lookaside buffer (TLB)
- Multi-level page table
- Track free memory: bitmaps or linked list
- Page replacement algorithms and modeling
- Working set of processes
- Other implementation issues
  - Dirty page and locking page etc.

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 physical frame # for one 2nd level page table
- 2nd level page table has actual physical frame numbers for the memory address

Why it is good?
- We don't have to allocate all levels initially, which reduces the size of page table
- They don't have to be continuous
Example: 2-level Address Translation

Page number
\[ p_1 = 10 \text{ bits} \]
\[ p_2 = 10 \text{ bits} \]
Page offset
\[ \text{offset} = 12 \text{ bits} \]

Page table base

Which tables should be in memory?

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, what is the size for the full page table?
  - Level-0: 1024 entries * 4bytes
  - Level-1: \( 1024 \times 1024 \) entries = 1M entries * 4bytes
  - Total: 4M + 4K

Page table size

- 32bit machine, page size 4k, each entry 4 bytes, one level page table (full 4GB linear address)
  
  \[ \text{Page table size} = 2^{20} \text{ pages} = 2^{22} = 4M \]

- 32bit machine, page size 4k, each entry 4 bytes, two level page table (two pages: 0x00000000, and 0xFFFFF000)
  
  \[ \text{Page table size} = (2^{10} \text{ level-0 entries}) \times 4 \text{bytes} + (2^{10} \text{ level-1 entries} \times 4 \text{bytes}) \times 2 = 12 \text{ Kbytes} \]

Memory Requirement of Page Tables

- Example: a process access 32 MB (recall that 1GB memory and 32 bits virtual address), what is the minimum and maximum memory for page table?
  - 4KByte / page \( \Rightarrow \) process has at most 8K virtual pages
  - One 2nd level page table maps 210 pages.

- Computing the minimum memory consumption
  - Minimum number of 2-level page entries needed (all pages are continuous):
    \[ 8 \text{K Virtual pages} \times 2^{10} = 8 \]
  - Total (minimum) memory for page table: 1st level page table + 8; in total, we need 9 pages to hold page tables
    \[ 9 \times 4 \text{KB} = 36 \text{KB} \]
Memory Requirement of Page Tables

- Example: a process accesses 32 MB (recall that 1GB memory and 32 bits virtual address), what is the minimum and maximum memory for page table
  - 4KB/page → process has at most 8K virtual pages
  - One 2nd level page table maps 210 pages;

- Computing the maximum memory consumption
  - 8K or more virtual pages will spread to all 2nd level page tables, which only has 1024 pages in total
  - Thus, in total, we will have 1 page for 1st level page table + 1024 pages for 2nd level. Thus, we will have the maximum as 4M+4K

Quiz

- Why the page table has to be physically continuous?
- If we have two pages (0x00000000 and 0x00201000), what is the size of page table?
- What are the tradeoffs to have small page size and large page size?
- What are the advantages and disadvantages of using a single-level page table?

Fragmentation

- **External Fragmentation**
  - total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation**
  - allocated memory larger than requested; this size difference is called internal partition.
- How can we reduce external fragmentation
  - **Compaction**: migrate memory contents to place all free memory together in one large block
  - Compaction is possible only if relocation is dynamic, and is done at execution time
Paging: Internal Fragmentation

- Calculating internal fragmentation
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - 35 pages + 1,086 bytes
  - Internal fragmentation of 2,048 - 1,086 = 962 bytes
  - Worst case fragmentation = 1 frame – 1 byte
  - On average fragmentation = 1 / 2 frame size
- So small frame sizes desirable? → more entries
- Page sizes growing over time
  - Solaris supports two page sizes – 8 KB and 4 MB

Size of Page/Frame: How Big?

- Determined by number of bits in offset (12Bit → 4KB)
- Smaller pages have advantages
  - Less internal fragmentation
  - Better fit for various data structures, code sections
- Larger pages are better because
  - Less overhead to keep track of them
  - More efficient to transfer larger pages to and from disk
- One principle: all entries of one-level page table → fit into one frame

32bits machine, 10 bits for each level

Designing of Multi-level Page Table

- Suppose that a system has a 28-bit logical address space and is byte-addressable. The amount of physical memory is 1MB (i.e., the 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 4 bytes (larger than 20 bits)
  2. One page can hold 256 entries (1K/4=256)
  3. Thus, we need 8 bits for the 2nd level page table

Designing of Multi-level Page Table

- Suppose that a system has a 28-bit logical address space and is byte-addressable. The amount of physical memory is 1MB (i.e., the physical address has 20 bits) and the size of a page/frame is 1K bytes. What about this design?

<table>
<thead>
<tr>
<th>8 bits</th>
<th>10 bits</th>
<th>10 bits</th>
</tr>
</thead>
</table>

Second level 10 bits, can’t be fitted into one page. Then we may need to have multiple pages (4 pages) that are continuous, which can’t guarantee or increase the complexity of OS design
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 is the benefit to use 3-level page table?
What is the shortcoming?

Benefits:
1. Reduce memory consumptions
2. Support different architectures, x86: 2 levels, SPARC: 3-levels. Easily collapse

Problems:
1. Expensive looking up

How can we make the address translation faster?

Translation Lookaside Buffer (TLB)

- Small Hardware: fast
- Store recent accessed mapping of page \( \rightarrow \) frame (64 \( \sim \) 1024 entries)
- If desired logical page number is found, get frame number from TLB
- If not:
  1. Get frame number from page table in memory
  2. Use standard cache techniques
  3. Replace an entry in the TLB with the logical & physical page numbers from this reference
  4. Contains complete page table entries for small number of pages

Address Translation with TLB

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

- Most caches are "physically addressed"
  - Accessed by physical addresses
  - Allows multiple processes to have blocks in cache at same time (otherwise context switch == cache flush)
  - Allows multiple processes to share pages
  - Cache doesn’t need to be concerned with protection issues
  - Access rights checked as part of address translation
- Perform address translation before cache lookup
  - Could involve memory access itself (to get PTE)
  - So page table entries can also be cached

Integrating TLB and Cache

- "Translation Lookaside Buffer" (TLB)

Basic overflow

1. CPU will generate an virtual address
2. Then we will check the TLB to find the mapping of this page. If the corresponding entry exists, then we could perform the translation and get the PA address. Otherwise, we will go through the slow address translation, and then save the mapping into TLB (typically after the translation).
3. After getting the PA address, we can check whether the corresponding cache line is in the cache or not. If yes, then we will have a cache hit and return the memory unit back to CPU.
4. Otherwise, it is a cache miss. We will fetch the cache line into the cache, and then return the memory unit back to 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 1 memory reference

CPU

Translation

hit

Cache

miss

Main Memory

VA

PA

hit

data

CPU

TLB Lookup

Cache

Main Memory

VA

PA

hit

data
Example of Memory Access Time

Assuming TLB has an access time of 4 ns and the memory access time is 20 ns. The disk access time is 8 ms. Page/frame is 1K bytes.

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

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)
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 = 8.2 microseconds. This is a slowdown by a factor of 40!!
- If we want just 10% performance degradation, then $p$ should be $220 > (1 - p) \times 200 + p \times (8 \text{ milliseconds})$
  - $< 0.00000025$, i.e., 1 page fault out of 400,000 accesses

THRASHING

Processes are busy swapping pages in and out
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. 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
- Process migrates from one locality to another
- Localities may overlap
- When the size of locality > total memory size, thrashing occurs...

Working-Set Model

- Informal Definition: the collection of pages that a process is working with, 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 Δ, the working set parameter. At any given time, all pages referenced by a process in its last Δ seconds of execution are considered to comprise its working set.
  - Pages outside the working set may be discarded at any time.

Working-Set Definition

- Δ = working-set window = a fixed number of page references, example: 10,000 instruction
- WSS (working set of Process P) = total number of pages referenced in the most recent Δ
  - if Δ too small, will not encompass entire locality
  - if Δ too large, will encompass localities
  - if Δ = ∞ ⇒ will encompass entire program
- D = Σ WSS = total demand frames
  - if D > (available frames) m ⇒ Thrashing
  - Thus, if D > m, then suspend one of the processes
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 reference bit and in-memory bit for each page
  - At a timer interrupt, copy and set 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?
- Improve = 10 bits and interrupt every 1000 time units

Balance Set

- Working set is not enough to control thrashing
- If the sum of the working sets of all runnable processes is greater than the size of memory, then refuse to run some of the processes
- Divide runnable processes up into two groups: active and inactive.
  - When a process is made active its working set is loaded, when it is made inactive its working set is allowed to migrate back to disk.
  - The collection of active processes is called the balance set

Page-Fault Frequency (PFF) Scheme

- Working set is a clumsy way to control thrashing
- PFF is a more direct way
  - High PFF $\Rightarrow$ more thrashing
  - Establish "acceptable" page-fault rate
    - If actual rate is too low, process loses frame
    - If actual rate is too high, process gains frame
  - Suspend a process if PFF is above upper bound and there is no free frames!

What happens if there is no free frame?
- Terminate a user program
- Swap out some page

PAGE REPLACEMENT
Page Replacement

- No free frame 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 improvement, 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.
- Restart 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.

  reference string:

  7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
**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.
- How do you know the future?
- Used for measuring how well your algorithm performs.

**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.

**Problem of LRU:**

- How to implement it efficiently?
- Full LRU needs to sort all time of reference.

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?
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 performance

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

Reference bit
- With each page associate a reference bit, initially = 0
- When page is referenced, set this bit to 1 by hardware
- Replace the one which is 0 (if one exists)
  - We do not know the order, however
  - Additional bits can help to gain more ordering information

Second chance Alg (uses one reference bit)
- FIFO with an inspection of ref bit
- If ref bit is 0,
  - replace that page
  - set its ref bit to 1
- If ref bit is 1, /* give a second chance */
  - set ref bit to 0
  - leave page in memory
  - Arrival time set to current time
  - go to next one
- Enhanced it modify bit, avoid replacing modified pages

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 (Optim)</td>
<td>Not implementable, but useful as a benchmark</td>
</tr>
</tbody>
</table>
ALLOCATING KERNEL MEMORY

Treated differently from user memory. Allocate 1 page even when 1 byte is needed.
Often allocated from a different free-memory pool.
Kernel requests memory for structures of varying sizes.
Some kernel memory needs to be contiguous.

Kernel Memory Allocation

- E.g., Linux PCB (struct task_struct)
  - > 1.7 Kbytes each
  - Created on every fork and every thread create (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 $2^n$.

Free Page’s List
**Page 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.

**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 $2^b$ into blocks of size $2^{b+1}$.
- Two blocks are buddies if:
  - They have equal size $2^b$.
  - They are located at contiguous physical addresses.
  - The address of the first page frame in the first block is aligned on a multiple of $2^{b+2}$.  
- 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`)
WHAT HAPPENS WHEN ALLOCATING MEMORY

### 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

### 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 33K) are usable. `ptr` is 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 memory will be managed by user space memory manager.
  - How to manage the memory inside user space?
Summary

- Simple memory management: swap etc.
- Virtual memory and paging
  - Page table and address translation
- Translation lookaside buffer (TLB)
- Multi-level page table
- Page replacement algorithms
- Working set of processes
- Kernel Memory Management