Virtual Memory

Operating Systems
Based on: Three Easy Pieces by Arpaci-Dusseaux

Moshe Sulamy

Tel-Aviv Academic College

Processes & Physical Addresses

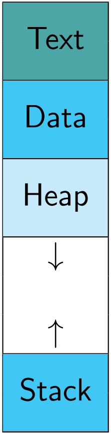
Can physical addresses be used while running? Yes. However:

- One process in memory at a time:
  - OS loads process to a fixed addresss? Cumbersone
  - OS loads at arbitrary place? Position independent code
- Several process in memory at a time:
  - Code must be position independent
  - Process can access other process memory
  - External fragmentation: Process cannot load.
    - Must wait for processes to terminate

> Physical addresses and ‘process’ addresses need to be separate

Process View of Memory

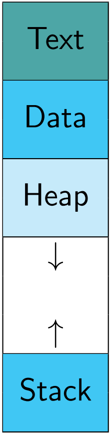
- 4GB (32-bit)
  - Not really. Linux takes top 1GB. Windows top 2GB.
- Heap, Stack, Data, Code (Text)



Process View of Memory

- 4GB (32-bit)
  - Not really. Linux takes top 1GB. Windows top 2GB.
- Heap, Stack, Data, Code (Text)

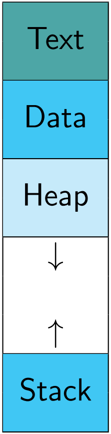
- Heap
  - Dynamic memory allocation
- Stack
  - Automatic memory allocation



Process View of Memory

- 4GB (32-bit)
  - Not really. Linux takes top 1GB. Windows top 2GB.
- Heap, Stack, Data, Code (Text)

- Heap
  - Dynamic memory allocation
- Stack
  - Automatic memory allocation
- Data
  - Static (Global/Local) values and variables
- Code
  - Program instructions



Processeses vs. Memory

- Each process has its own **address space**
  - 4GB (32-bit)
- Ten processes: 40GB of RAM!
  - Typically we have much less physical memory
  - And many more processes

> How can the OS provide a private, potentially large address space for multiple running processes?

Virtualizing Memory

- OS and hardware virtualizes physical memory
- Goals:
  - **Transparency**:
    - Invisible to the running program
    - A process "thinks" it has a continuous **address space**
  - **Efficiency**:
    - Not making programs run much more slowly
    - Not using too much memory to support virtualization
  - **Protection**:
    - Protect processes from one another, and the OS from processes
    - **Isolation** among processes

Address Translation

- **Hardware-based address translation**
  - On every memory reference, address translation is performed
  - Hardware redirects memory references to physical locations
- OS manages memory locations
  - Which are free and which are in use

- Hardware support
  - e.g., registers, TLBs, page-table

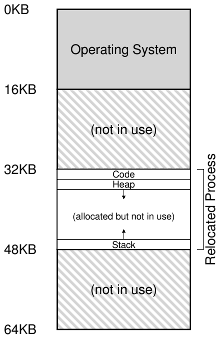
Assumptions

1. Address space must be placed contiguously
2. The size is less than the physical memory size
3. Each address space is the same size

Dynamic Relocation

Also called: **base and bounds**

- Hardware registers: **base** and **bounds**
- OS decides where in physical memory a process is loaded
  - Sets **base** register to that value
  - Sets **bounds** register
- Memory references are **translated** by MMU
  - `physical address = virtual address + base`
- MMU checks reference is within **bounds** of base



Hardware Support

- Two (or more) CPU modes:
  - OS runs in **privileged mode** (or **kernel mode**)
  - Applications run in **user mode**
- **Base** and **bounds** registers
  - Hardware is called **memory management unit (MMU)**
- Generate **exceptions** on illegal access
  - Execute OS **exception handler**

We are in better situation

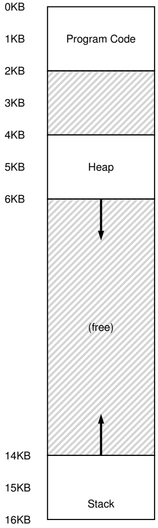
- Solved cross process memory access
- External fragmentation exists but situation is better
  - compactification (not reasonable for huge memories)
- Code can believe it is loaded in the same address always
- Simple and efficient.

Same issues as before relocation

- A process starts running:
  - Find space for address space in physical memory
  - Maintain a **free list** of free address spaces
- A process is terminated:
  - Reclaim the memory for use
  - Add it back to the **free list**

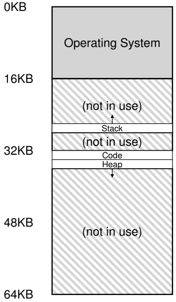
Segmentation

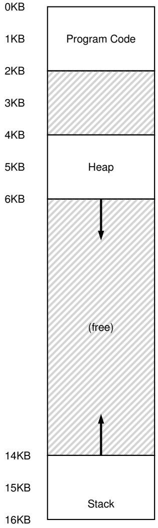
- Base and bounds is problematic
  - Big chunk of free space
  - Still taking up continous physical memory
- Solution?



Segmentation

- Base and bounds is problematic
  - Big chunk of free space
  - Still taking up continous physical memory
- Solution? **segmentation**
  - Base and bounds pair for each segment





Segmentation

- A contiguous portion of the address space
  - Logical segments: code, stack, heap
- In essence each segment has its own address space
- Each can be placed in a different part of the physical memory
- Usually under program (or rather compiler) control
- `physical address = offset + base`
  - Not `virtual address + base`!
  - e.g., offset of virtual address 100 is `100`
  - Offset of virtual address 4200 can be `104`
    - Since it is `104` in the heap segment

Segmentation

- Ever encountered a **segmentation fault**?
- If an **illegal address** beyond the segment is referenced:
  - Hardware detects **out of bounds** access
  - OS event: **segmentation fault**

Support for Sharing

- Segment can be shared between address spaces (processes)
  - **Code sharing** is common
  - Same program, no need to load the code twice
- Extra hardware support: **protection bits**
  - Code segment is read-only
  - Can be shared without harming **isolation**

Segment	Base	Size	Grows Positive?	Protection
Code	32K	2K	1	Read-Execute
Heap	34K	3K	1	Read-Write
Stack	28K	2K	0	Read-Write

Fine-Grained vs. Coarse-Grained

- Thus far: just a few segments
  - Code, stack, heap
  - **Coarse-grained**
- **Fine-grained**
  - Large number of smaller segments
  - **Segment table** stored in memory
  - More flexible

OS Support

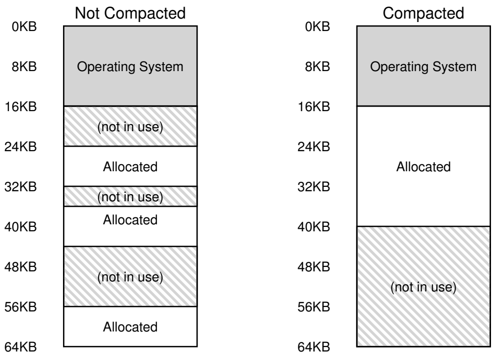
- On memory allocation/free: update segment size

OS Support

- On memory allocation/free: update segment size
- **External fragmentation**
  - Physical memory becomes full of little holes of free space
  - Difficult to allocate new segments, or grow existing ones
    - But possible: Compactification

Compaction

- **Compaction**
  - Stop running processes
  - Copy used memory to contiguous region
  - Change segment registers accordingly
  - Compaction is expensive!



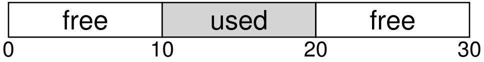
General Segmentation Note

Dynamic relocation and segmentation are relic of the past

Detour - Heap management

External Fragmentation

- Detour to discuss **free-space management**
- Also applies to user-level memory allocation
    - e.g., `malloc()` and `free()`
- Not a problem with fixed-size chunks
    - Can use a free-list



20 free bytes fragmented into chunks of 10

External Fragmentation

Assume:

- A basic `heap` interface:
  - `malloc(size_t size)` allocates `size` or more bytes
  - `free(void *ptr)` frees corresponding chunk
    - Note that no size is provided
- Only **external fragmentation**
  - Allocators also have **internal fragmentation**
  - Unused space in chunks bigger than requested
- No memory relocation
  - No **compaction** of free space

Splitting

Splitting

`malloc(2048)`

Splitting

`malloc(2048)`

Splitting

`malloc(2048)=15KB`

Splitting

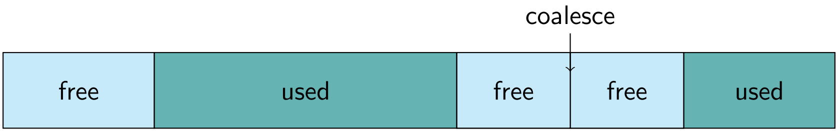
Coalescing

`free(15KB)`

Coalescing

`free(15KB)`

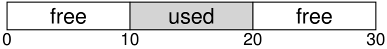
Coalescing



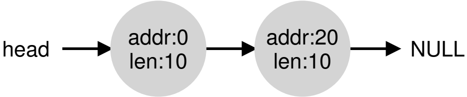
Coalescing

Splitting

- Find a free chunk to satisfy request and split it into two
  - Assume the following 30-byte heap:



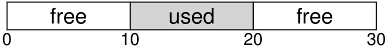
  - Its free list is:



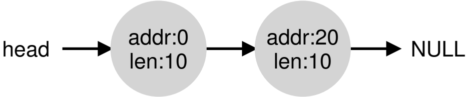
  - After a 1-byte request:

Splitting

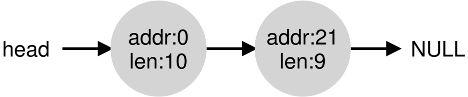
- Find a free chunk to satisfy request and split it into two
  - Assume the following 30-byte heap:



  - Its free list is:

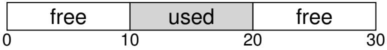


  - After a 1-byte request:

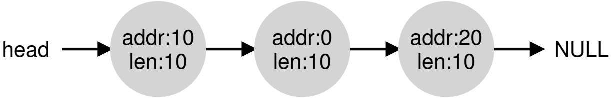


Coalescing

- Coalesce free space when memory is freed
  - i.e., merge contiguous free chunk
  - Consider our previous heap:

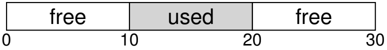


  - After a call to `free(10)`:

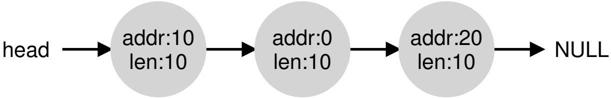


  - With coalescing:

- Coalesce free space when memory is freed
  - i.e., merge contiguous free chunk
  - Consider our previous heap:



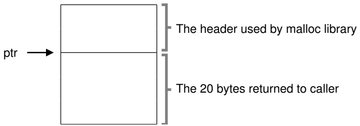
  - After a call to `free(10)`:

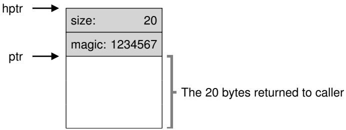


  - With coalescing:

Tracking The Size

- Interface to `free(void *ptr)` does not provide size
- Store extra information in a **header** block
  - Usually just before chunk of memory
  - Magic number for integrity checking



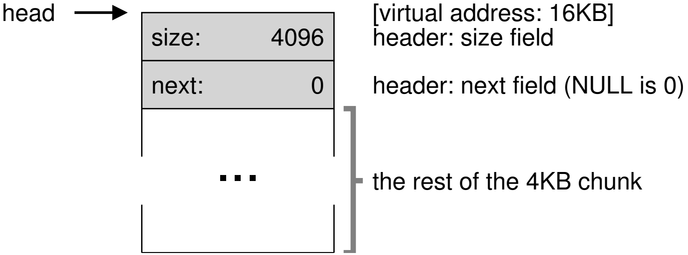


Free List

- Need to implement the **free list** itself
  - Can't call `malloc()` - we are implementing it!
  - Need to embed the list inside the free space itself

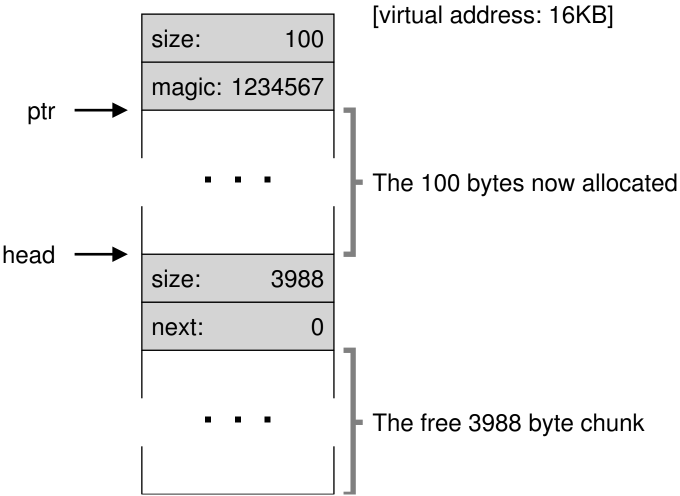
Free List

- Need to implement the **free list** itself
  - Can't call `malloc()` - we are implementing it!
  - Need to embed the list inside the free space itself
- Example: manage 4096-byte chunk
  - Maintain `size` and `next` for each node:



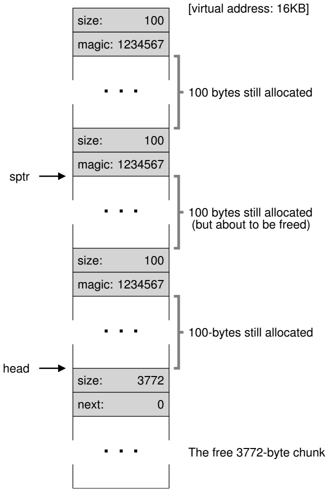
Free List

- 100 bytes are requested: **split** chunk



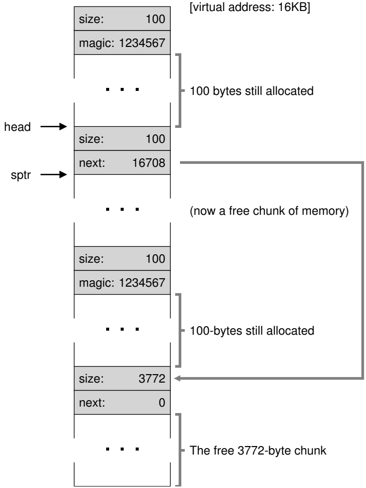
Free List

- Three allocated regions of 100 bytes:



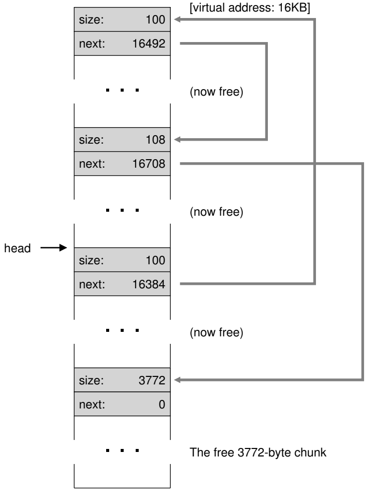
Free List

- After `free(16500)`:
  - Region start: 16384, previous chunk: 108, current header: 8



Free List

- Free last two chunks: fragmentation!
  - Need to **coalesce** the list



Managing Free Space

- **Best Fit**
  - Return smallest chunk that's as big or bigger than requested size
  - Exhaustive search: heavy performance penalty
- **Worst Fit**
  - Opposite of best fit: return largest chunk
  - Still requires full search, bad performance, excess fragmentation

Managing Free Space

- **First Fit**
  - Return first block that is big enough
  - Speed advantage, but pollutes beginning of list
- **Next Fit**
  - As first fit, but start where stopped previously
  - Spreads the searches throughout the list

Managing Free Space

- Examples: allocation request size 15

  head → 10 → 30 → 20 → NULL

- Best-fit:

  head → 10 → 30 → 5 → NULL

- Worst-fit:

  head → 10 → 15 → 20 → NULL

- First-fit: same as worst-fit, but faster

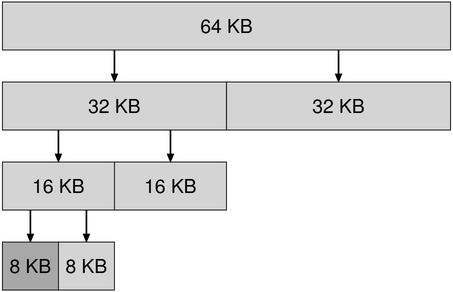
Segregated Lists

- Keep lists for fixed-size objects
- General memory allocator for the rest

- **Slab allocator** allocates **object caches**
    - For common kernel objects (locks, file-system inodes, etc.)
    - When a cache is running low: request **slab** of memory from general allocator

Buddy Allocation

- Make coalescing simple: **binary buddy allocator**
- Divide free space by two until a block is found
  - Further split into two is too small
  - Suffers from **internal fragmentation**
  - Easy to coalesce:
    - Recursively up the the tree
    - Buddy address differs by a single bit



Summary

- Virtualize RAM into process **address space**
- Address translation:
  - Dynamic relocation (**base and bounds**)
  - Segmentation
    - **Coarse-grained**: just a few segments
    - **Fine-grained**: large number of smaller segments
- **External fragmentation**
  - Free memory fragments into small parts
  - Splitting & coalescing
- **Compaction**
  - Stop processes, copy data to contiguous region