Virtual Memory and Dynamic Memory Allocation
A Few Problems with Physical Addressing...
Problem 1: Memory Management

What goes where?

Process 1
Process 2
Process 3
...
Process n

stack
heap
code
globals
...

Main memory

Also:
Context switches must swap out entire memory contents. Isn't that expensive?
Problem 2: Capacity

64-bit addresses can address several exabytes
(18,446,744,073,709,551,616 bytes)

Physical main memory offers a few gigabytes
(e.g. 8,589,934,592 bytes)

(To scale with 64-bit address space, you can't see it.)

1 virtual address space per process, with many processes...
Problem 3: Protection

Problem 4: Sharing
Solution: Virtual Memory (address indirection)

Private virtual address space per process.

Single physical address space managed by OS/hardware.

Virtual-to-physical mapping
Physical addresses are *invisible* to programs.

Virtual Addressing and Address Translation

Memory Management Unit
translates virtual address to physical address

CPU Chip

Virtual address (VA) 4100

MMU

Physical address (PA) 4

Main memory

Data

Physical addresses are *invisible* to programs.
Page-based Mapping

fixed-size, aligned pages
page size = power of two

Map virtual pages onto physical pages.

Some virtual pages do not fit!
Where are they stored?
array of *page table entries* (PTEs) mapping virtual page to where it is stored

---

**How many page tables are in the system?**

---

Memory resident, managed by HW (MMU), OS
Address Translation with a Page Table

Virtual address (VA)

Virtual page number (VPN)
Virtual page offset (VPO)

Page table

Base address of current process's page table

Valid Physical page number (PPN)

Physical page number (PPN)
Physical page offset (PPO)

Physical address (PA)

Virtual page mapped to physical page?
Yes = Page Hit
Page Hit: virtual page in memory
Page Fault:

<table>
<thead>
<tr>
<th>Virtual Page Number</th>
<th>Physical Page Number or disk address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td>PP 0</td>
</tr>
<tr>
<td>1</td>
<td>PP 1</td>
</tr>
<tr>
<td><strong>0</strong></td>
<td><strong>On disk</strong></td>
</tr>
<tr>
<td>1</td>
<td>PP 3</td>
</tr>
<tr>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>0</td>
<td>On disk</td>
</tr>
<tr>
<td>1</td>
<td>PP 2</td>
</tr>
</tbody>
</table>

Page table

Physical pages (Physical memory)

- VP 1
- PP 0
- VP 2
- PP 1
- VP 7
- PP 2
- VP 4
- PP 3

Swap space (Disk)

- VP 3
- VP 6
Page Fault: exceptional control flow

Process accessed virtual address in a page that is not in physical memory.

User Code

OS exception handler

movl exception: page fault

Load page into memory

return

Returns to faulting instruction: movl is executed again!
Page Fault: 1. page not in memory

Virtual Page Number

Physical Page Number
or disk address

Valid

0 null
1 PP 0
1 PP 1
0 On disk
0 null
0 On disk
1 PP 2

PTE 0

PTE 7

Physical pages
(Physical memory)

VP 1
VP 2
VP 7
VP 4

Swap space
(Disk)

VP 3
VP 6

What now?

OS handles fault
Page Fault: 2. OS evicts another page.

Virtual Page Number

Physical Page Number or disk address

Valid

<table>
<thead>
<tr>
<th>PTE 0</th>
<th>0</th>
<th>null</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>On disk</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>PP 1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>On disk</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>PP 3</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>On disk</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>PP 2</td>
<td></td>
</tr>
</tbody>
</table>

Swap space (Disk)

VP 3

VP 6

VP 1

Physical pages (Physical memory)

VP 1

VP 2

VP 7

VP 4

"Page out"
Page Fault: 3. OS loads needed page.

Finally:
Re-execute faulting instruction.
Page hit!
Dynamic
Memory
Allocation
## Heap Allocation

<table>
<thead>
<tr>
<th>Addr</th>
<th>Perm</th>
<th>Contents</th>
<th>Managed by</th>
<th>Initialized</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>R</td>
<td>String literals</td>
<td>Compiler/Assembler/Linker</td>
<td>Startup</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Instructions</td>
<td>Compiler/Assembler/Linker</td>
<td>Startup</td>
</tr>
<tr>
<td>2^{N-1}</td>
<td>RW</td>
<td>Procedure context</td>
<td>Compiler</td>
<td>Run-time</td>
</tr>
<tr>
<td></td>
<td>RW</td>
<td>Dynamic data structures</td>
<td>Programmer, malloc/free, new/GC</td>
<td>Run-time</td>
</tr>
<tr>
<td></td>
<td>RW</td>
<td>Global variables/static data structures</td>
<td>Compiler/Assembler/Linker</td>
<td>Startup</td>
</tr>
</tbody>
</table>

**Legend:**
- **RW**: Read/Write
- **R**: Read Only
- **X**: Executable
Pages too coarse-grained for allocating individual objects. Instead: flexible-sized, word-aligned blocks.

void* malloc(size_t size);

void free(void* ptr);
Example (64-bit words)

p1 = malloc(32);

p2 = malloc(40);

p3 = malloc(48);

free(p2);

p4 = malloc(16);
Allocator Goals: malloc/free

1. Programmer does not decide locations of distinct objects.
   - Programmer decides: what size, when needed, when no longer needed

2. Fast allocation.
   - mallocs/second or bytes malloc'd/second

3. High memory utilization.
   - Most of heap contains necessary program data.
   - Little wasted space.

   - Enemy: fragmentation – unused memory that cannot be allocated.
Internal Fragmentation

- payload smaller than block

Causes
- metadata
- alignment
- policy decisions
External Fragmentation (64-bit words)

Total free space large enough,

but no contiguous free block large enough

\[ p1 = \text{malloc}(32); \]
\[ p2 = \text{malloc}(40); \]
\[ p3 = \text{malloc}(48); \]
\[ \text{free}(p2); \]
\[ p4 = \text{malloc}(48); \]

Depends on the pattern of future requests.
Implementation Issues

1. Determine **how much to free** given just a pointer.

2. **Keep track of free blocks.**

3. **Pick a block to allocate.**

4. Choose what do with **extra space when allocating** a structure that is **smaller than the free block used.**

5. **Make a freed block available for future reuse.**
Knowing How Much to Free

- Keep length of block in *header* word preceding block

```
p0 = malloc(32);
p0 = malloc(32);
```

Takes extra space!

- Takes extra space!
Keeping Track of Free Blocks

Method 1: *Implicit list* of all blocks using length

Method 2: *Explicit list* of free blocks using pointers

Method 3: *Seglist*
Different free lists for different size blocks

More methods that we will skip...
**Implicit Free List: Block Format**

**Block metadata:**
1. Block size
2. Allocation status

*Store in one header word.*

**payload**
(application data, when allocated)

1 word

**optional padding**

Steal LSB for status flag.
LSB = 1: allocated
LSB = 0: free

16-byte aligned sizes have 4 zeroes in low-order bits

00000000
00010000
00100000
00110000
...

---
Implicit Free List: Heap Layout

Start of heap

Block Header (metadata)
block size | block allocated?

Payloads start at 16-byte (2-word) alignment.
Blocks sizes are multiples of 16 bytes.

Special end-heap word
Looks like header of zero-size allocate block.

Initial word can't be part of block.
May force internal fragmentation.

Free word
Allocated word
Allocated word wasted
Implicit Free List: Finding a Free Block

First fit:
Search list from beginning, choose \textit{first} free block that fits

Next fit:
Do first-fit starting where previous search finished

Best fit:
Search the list, choose the \textit{best} free block: fits, with fewest bytes left over
Implicit Free List: **Allocating a Free Block**

```c
p = malloc(24);
```

Allocated space $\leq$ free space.
Use it all? Split it up?

Now showing allocation status flag implicitly with shading.
Implicit Free List: **Freeing a Block**

```c
free(p);
```

Clear *allocated* flag.

```c
malloc(40);
```

**External fragmentation!**

Enough space, not one block.
**Coalescing Free Blocks**

\[ \text{free}(p) \quad \textbf{Coalesce} \quad \text{with following free block.} \]

\[ \begin{array}{ccccccc}
32 & 32 & 16 & 16 \\
\end{array} \]

\[ \text{Coalesce with preceding free block?} \]

\[ \begin{array}{ccccccc}
32 & 32 & 48 & 16 & 16 \\
\end{array} \]

\[ \text{logically gone} \]
Bidirectional Coalescing: Boundary Tags

Header

payload
(application data, when allocated)

Boundary tag
(footer)

32 32 32 32 48

48 32 32

[Knuth73]
### Constant-Time Coalescing: 4 cases

<table>
<thead>
<tr>
<th>Case 1</th>
<th>Case 2</th>
<th>Case 3</th>
<th>Case 4</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Diagram 1" /></td>
<td><img src="image2.png" alt="Diagram 2" /></td>
<td><img src="image3.png" alt="Diagram 3" /></td>
<td><img src="image4.png" alt="Diagram 4" /></td>
</tr>
</tbody>
</table>

**Case 1:**
- Freed Block: m1
- n
- m2

**Case 2:**
- Freed Block: m1
- n
- m2

**Case 3:**
- Freed Block: m1
- n
- m2

**Case 4:**
- Freed Block: m1
- n
- m2

**Notes:**
- In each case, the freed blocks are coalesced in constant time.
- The diagrams illustrate the movement and merging of blocks to optimize memory usage.
Summary: Implicit Free Lists

Implementation: simple
Allocate: $O(\text{blocks in heap})$
Free: $O(1)$

Memory utilization: depends on placement policy

Not widely used in practice
some special purpose applications

Splitting, boundary tags, coalescing are general to all allocators.
Explicit Free Lists

**Allocated block:**

- **payload** *(application data, when allocated)*
- optional padding
- **block size**

**Free block:**

- **block size**
- **next** pointer
- **prev** pointer
- **block size**

(same as implicit free list)

Explicit list of *free* blocks rather than implicit list of *all* blocks.
Explicit Free Lists: **List vs. Memory Order**

- Abstractly: doubly-linked lists
- Concretely: free list blocks in any memory order

**List Order ≠ Memory Order**
Explicit Free Lists: Allocating a Free Block

Before

After (with splitting)

= malloc(...)
Explicit Free Lists: **Freeing a Block**

*Insertion policy:* Where in the free list do you add a freed block?

**LIFO (last-in-first-out) policy**
- **Pro:** simple and constant time
- **Con:** studies suggest fragmentation is worse than address ordered

**Address-ordered policy**
- **Con:** linear-time search to insert freed blocks
- **Pro:** studies suggest fragmentation is lower than LIFO

LIFO Example: 4 cases of freed block neighbor status.
Freeing with LIFO Policy: between allocated blocks

Before

Insert the freed block at head of free list.

After
Freeing with LIFO Policy: between free and allocated

Before

Head

Splice out predecessor block, coalesce both memory blocks, and insert the new block at the head of the free list.

After

Head

Could be on either or both sides...
Splice out successor block, coalesce both memory blocks and insert the new block at the head of the free list.
Freeing with LIFO Policy: between free blocks

Before

Splice out predecessor and successor blocks, coalesce all 3 memory blocks and insert the new block at the head of the list.

After
Summary: Explicit Free Lists

Implementation: fairly simple

Allocate: \(O(\text{free blocks})\) vs. \(O(\text{all blocks})\)
Free: \(O(1)\) vs. \(O(1)\)

Memory utilization:
depends on placement policy
larger minimum block size (next/prev) vs. implicit list

Used widely in practice, often with more optimizations.

Splitting, boundary tags, coalescing are general to all allocators.
Seglist Allocators

Each *size bracket* has its own free list

Faster best-fit allocation...
Summary: **Allocator Policies**

All policies offer trade-offs in fragmentation and throughput.

**Placement policy:**
- First-fit, next-fit, best-fit, etc.
- *Seglists* approximate best-fit in low time

**Splitting policy:**
- Always? Sometimes? Size bound?

**Coalescing policy:**
- Immediate vs. deferred
Remembrallocator Block Format

Allocated block:

<table>
<thead>
<tr>
<th>block size</th>
<th>p 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>payload</td>
<td></td>
</tr>
</tbody>
</table>

Free block:

<table>
<thead>
<tr>
<th>block size</th>
<th>p 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>next pointer</td>
<td></td>
</tr>
<tr>
<td>prev pointer</td>
<td></td>
</tr>
<tr>
<td>payload</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>block size</th>
<th>0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>payload</td>
<td></td>
</tr>
</tbody>
</table>

Minimum block size?
- Implicit free list
- Explicit free list

Update 2 headers on each malloc/free.