Virtual Memory and Memory Management

David Marchant
Based on slides by Troels Henriksen

2022-10-04

Inspired by slides by Randal E. Bryant and David R. O'Hallaron.

Simple memory system example

The Linux virtual memory system

Memory mapping

Dynamic allocation

Addressing

- 14-bit virtual addresses.
- 12-bit physical addresses.
- $64 = 2^6$ byte pages.

Virtual addresses

13	12	11	10	9	8	7	6	5	4	3	2	1	0
			VPN							VPO			

Physical addresses

11	10	9	8	7	6	5	4	3	2	1	0
		PPN						PPO			

TLB (4 sets, 4-way set associative)

Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	00	0	09	0D	1	00	0A	0	07	02	1
1	03	2D	1	02	10	0	04	0B	0	0A	03	0
2	02	3D	0	08	0F	0	06	05	0	03	2D	0
3	07	12	0	03	0D	1	0A	34	1	02	03	0

Page table

VPN	PPN	Valid	VPN	PPN	Valid	VPN	PPN	Valid	VPN	PPN	Valid
00	28	1	04	10	0	08	13	1	0C	13	0
01	27	0	05	16	1	09	17	1	0D	2D	1
02	33	1	06	28	0	0A	09	1	0E	11	1
03	02	1	07	28	0	0B	01	0	0F	0D	0

Cache (16 lines, direct mapped, physically addressed)

CT						CI				CO	
11	10	9	8	7	6	5	4	3	2	1	0
PPN						PPO					

Contents

Idx	Tag	Valid	Block	Idx	Tag	Valid	Block
0	19	1	99 11 23 11	**8**	24	1	3A 00 51 89
1	15	0	01 02 11 99	**9**	2D	0	89 00 3A 51
2	18	1	00 02 04 08	**A**	2D	1	93 15 DA 3B
3	36	0	11 23 11 99	**B**	08	0	89 51 15 3A
4	32	1	43 6D 8f 09	**C**	12	0	3A 00 15 69
5	0D	1	36 72 F0 1D	**D**	16	1	04 96 34 15
6	31	0	72 1D F0 36	**E**	13	1	83 77 1B D3
7	16	1	11 C2 DF 03	**F**	14	0	14 20 13 37

Translating virtual addresses and cache lookups

- **Virtual address:** `0x03D4`
  - VPN
    TLBI
    TLBT
    TLB Hit?
    Page fault?
    PPN
- **Physical address:**
  - CO
    CI
    CT
    Cache hit?
    Byte
- **Virtual address:** `0x0020`
  - VPN
    TLBI
    TLBT
    TLB Hit?
    Page fault?
    PPN
- **Physical address:**
  - CO
    CI
    CT
    Cache hit?
    Byte

Simple memory system example

The Linux virtual memory system

Memory mapping

Dynamic allocation

Structure of a page table

VPN	PPN	Valid	Flags
0			
1			
2			
3			
4			

⋮

- Showing a flat page table for simplicity.
- Actually a multi-level page table.

- Page table is accessed by MMU (hardware) so its structure is **fixed** and kept **simple**.
- Either a page is *there* or *it's not*.
  - No "page is on disk" information.
  - No "demand-paging" information.
  - If we access a non-valid page: **page fault**!
- Page faults are handled by **software** (kernel code), meaning we have **flexibility**.
  - Page fault *handler* can update the page table based on kernel data and policy.
- **But what does that actually look like?**

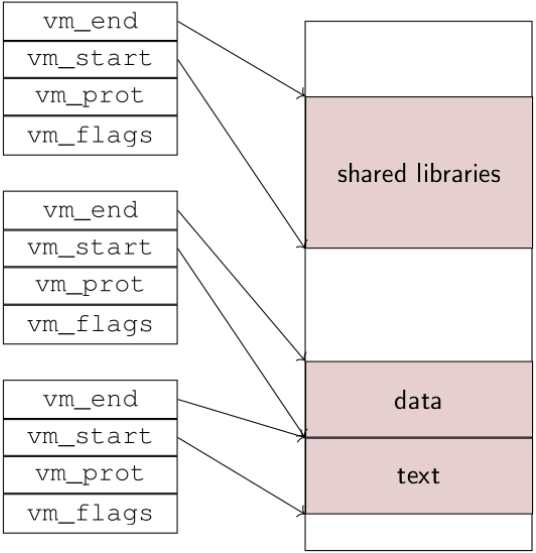
VM areas

Linux organises a virtual memory space as a set of *areas*.

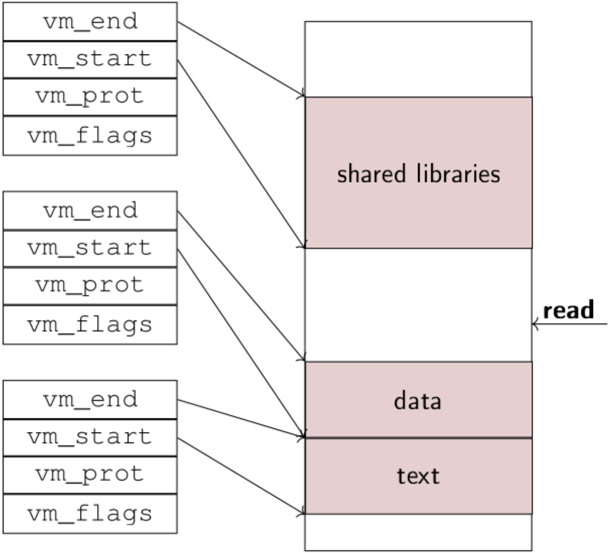
```
struct vm_area_struct {
        unsigned long vm_start;
        unsigned long vm_end;
        pgprot_t      vm_page_prot;
        ...
};
```

- Each area describes the properties of a span of virtual memory.
  - May cover any number of pages.
  - Area has uniform protection/access bits (read, write, exe).
- **CPU/MMU has no idea what this is**—purely a software data structyre.
  - Easier to change kernel code than hardware.

Handling page faults

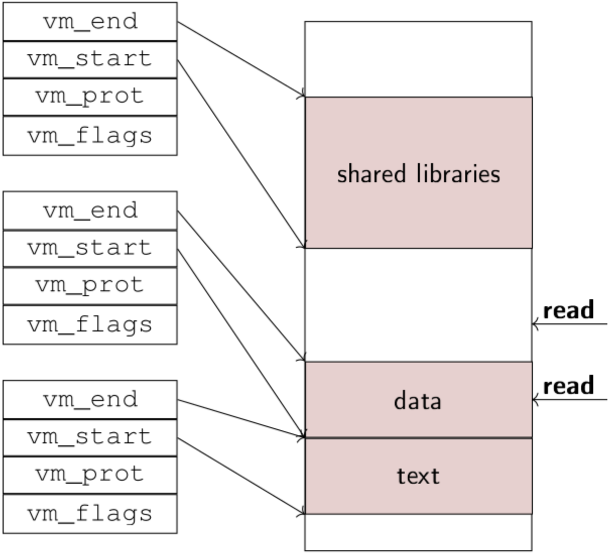


Handling page faults



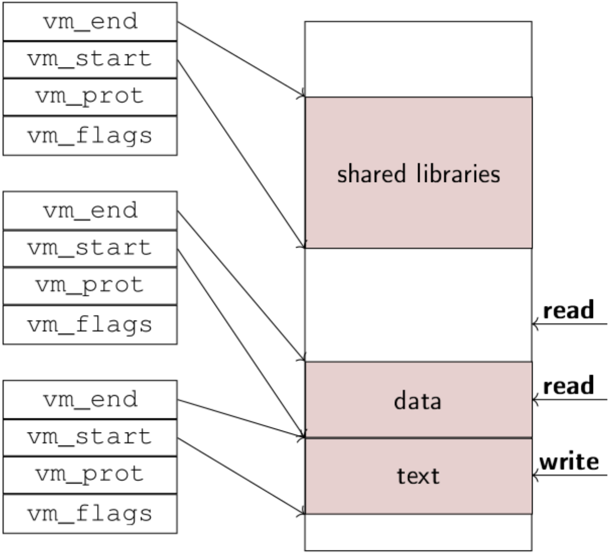
- **Segmentation fault**–page does not exist.

Handling page faults



- **Segmentation fault**–page does not exist.
- **Normal page fault**
  - Find available physical page in RAM.
  - Fetch page contents from disk.
  - Update page table.

Handling page faults



- **Segmentation fault**–page does not exist.
- **Normal page fault**
  - Find available physical page in RAM.
  - Fetch page contents from disk.
  - Update page table.
- **Segmentation fault**–permissions are wrong for type of access.

Simple memory system example

The Linux virtual memory system

Memory mapping

Dynamic allocation

Memory mapping

- **VM areas are associated with disk objects**
  - Known as *memory mapping*.
  - As a programming abstraction, lets us access a file as it were memory.
  - But also a strong organising principles.
- **Area can be *backed by* (get its initial values from)**:
  - **Regular file** on disk (e.g. a program executable file).
    - Initial page contents from from section of file.
  - **Anonymous file** (i.e. nothing).
    - Initial page contents are zero.
    - Once written to, like any other page.

Memory mapping

- **VM areas are associated with disk objects**
  - Known as *memory mapping*.
  - As a programming abstraction, lets us access a file as it were memory.
  - But also a strong organising principles.
- **Area can be *backed by* (get its initial values from)**:
  - **Regular file** on disk (e.g. a program executable file).
    - Initial page contents from from section of file.
  - **Anonymous file** (i.e. nothing).
    - Initial page contents are zero.
    - Once written to, like any other page.

```
struct vm_area_struct {
        ...
        unsigned long   vm_pgoff;
        struct file     *vm_file;
        ...
};
```

Memory mapping from userspace

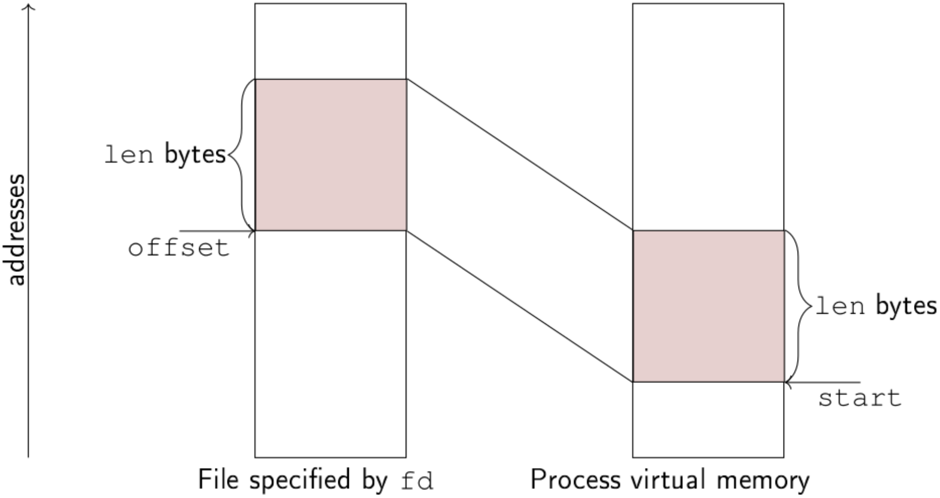
```
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

- Map `len` bytes starting at offset `offset` of the file descriptor `fd`, preferably at address addr .
  - addr may be NULL for "I don't care".
  - `prot`: `PROT_READ`, `PROT_WRITE`, `PROT_EXEC`, `PROT_NONE`.
  - `flags`: `MAP_ANONYMOUS`, `MAP_PRIVATE`, `MAP_SHARED`, many more...
- Returns a pointer to start of mapped area (may not be addr).

```
int munmap(void *addr, size_t length);
```

- Unmaps pages starting at this address.
  - VM area is cloven in twain if we unmap pages in the middle.

```c
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```



`mmap` examples

See lecture code.

Simple memory system example

The Linux virtual memory system

Memory mapping

Dynamic allocation

User-space allocators

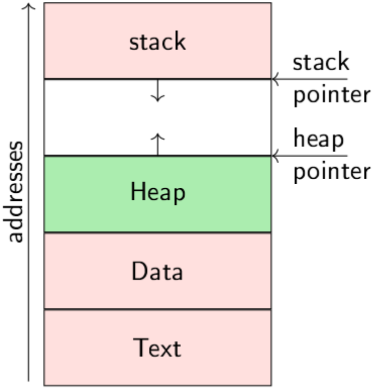
- `mmap()` is **powerful** but **inflexible**:
  - Smallest granularity of allocation is a page.
  - Is a system call, so fairly slow.

User-space allocators

- `mmap()` is **powerful** but **inflexible**:
  - Smallest granularity of allocation is a page.
  - Is a system call, so fairly slow.
- Instead programmers use **dynamic memory allocators** (e.g. `malloc()`) to aquire memory at runtime.
  - Run entirely in user space—not part of the kernel.
  - Acquires memory via `mmap()` and `sbrk()`.

User-space allocators

- `mmap()` is **powerful** but **inflexible**:
  - Smallest granularity of allocation is a page.
  - Is a system call, so fairly slow.
- Instead programmers use **dynamic memory allocators** (e.g. `malloc()`) to aquire memory at runtime.
  - Run entirely in user space—not part of the kernel.
  - Acquires memory via `mmap()` and `sbrk()`.
- Region of virtual memory managed by such an allocator is known as *the heap*.
  - No relation to the datastructure known as a heap.
  - May have multiple heaps; heap might not be contiguous.
  - When we say *the* heap, we mean whatever `malloc()` manages by default.



Dynamic memory allocation

- **Allocator maintains heap as collection of *blocks* of varying size, each of which is either *allocated* or *free*.**
- **Types of allocators:**

  Explicit allocator: application manually allocates and frees space.
  - E.g. `malloc()` in C.

  Implicit allocator: application allocates but freeing is automatic.
  - E.g. garbage collection in F#, SML, Haskell, Futhark, and Lisp.
- **CompSys discusses simple explicit allocators.**
  - **Implicit allocators** may be touched upon in PLD.

Allocator API

```
void *malloc(size_t size);
```

- Returns a pointer to a memory block of *at least* `size` bytes.
  - May round up to ensure address is always aligned.
- `NULL` on failure.

```
void free(void *ptr);
```

- Returns block of memory to heap.
- `p` must have been returned by `malloc()` (or one of the variants).

```
void *realloc(void *ptr, size_t size);
void *calloc(size_t nmemb, size_t size);
```

- Less crucial, but sometimes useful.

System-level building blocks

`malloc()` uses the following functions to request memory from the kernel.

```
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

- Use `MAP_ANONYMOUS` to map fresh pages (will be demand-paged when first accessed).

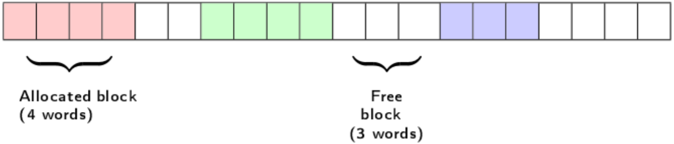
```
void *sbrk(intptr_t increment);
```

- Modifies *heap pointer* (traditionally called the *program break*).
- Maps or unmaps new anonymous read-write pages as necessary.
- Similar to growing the stack by decrementing stack pointer.
  - But with a *system call*: there is no register for the heap pointer.

This memory is then split into blocks and returned to callers of `malloc()` as needed.

Notation used in the following slides

- We show how the memory managed by `malloc` is split into blocks.
- We show 4-byte words instead of single bytes.



Free word

Allocated word (will use different colours to indicate different allocated blocks)

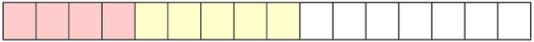
Allocation example

```
p1 = malloc(4*4);
```

Allocation example

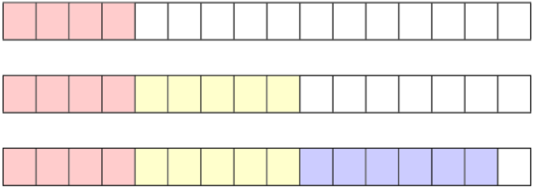
```
p1 = malloc(4*4);
```

```
p2 = malloc(5*4);
```



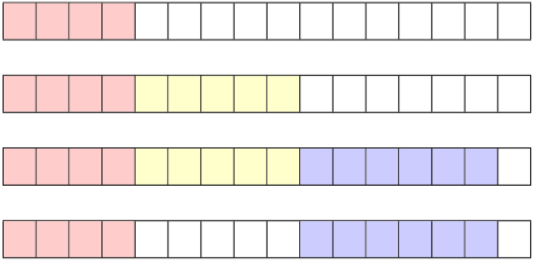
Allocation example

```
p1 = malloc(4*4);
p2 = malloc(5*4);
p3 = malloc(6*4);
```



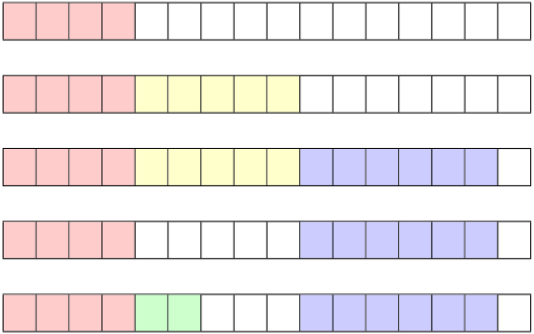
Allocation example

```
p1 = malloc(4*4);
p2 = malloc(5*4);
p3 = malloc(6*4);
free(p2);
```



Allocation example

```
p1 = malloc(4*4);
p2 = malloc(5*4);
p3 = malloc(6*4);
free(p2);
p4 = malloc(2*4);
```



Constraints

Applications

- Can issue arbitrary sequence of `malloc()` and `free()` requess.
- `free()` must be to a `malloc()`ed block.

Allocators

- Can't control number or size of allocated blocks.
- Blocks must be *contiguous* in memory.
- Must respond immediately to `malloc()` requests.
  - i.e. cannot reorder or delay requests.
- Must allocate blocks from free memory.
  - i.e. blocks cannot overlap.
- Can manipulate and modify only free memory.
  - When a block has been returned from `malloc()`, *that memory belongs to the application*.
- Cannot move blocks once `malloc()`d.
  - ...can someone say why?

Performance Goal I: *throughput*

- **We are given some sequence of `malloc` and `free` requests:**
  - $R_0, R_1, \ldots, R_{n-1}$.
- **Goal: maximise throughput**
- **Throughput:**
  - Number of completed requests per unit time.
    - E.g. suppose 5000 `malloc()`s and 5000 `free()`s in 10 seconds.
    - Throughput is 1000 ops/s.

Performance Goal II: *peak memory utilisation*

- We are given some sequence of **`malloc`** and **`free`** requests:
  - $R_0, R_1, \ldots, R_{n-1}$.
- **Goal: maximise memory utilisation**

Aggregate payload P_k:

- `malloc(p)` results in a block with a *payload* of p bytes.
- P_k is the sum of currently allocated payloads after request R_k.

Current heap size H_k:

- Total amount of heap memory requested from the system.
- Grows only when using `sbrk()` or `mmap()`.

Peak memory utilisation after k requests U_k:

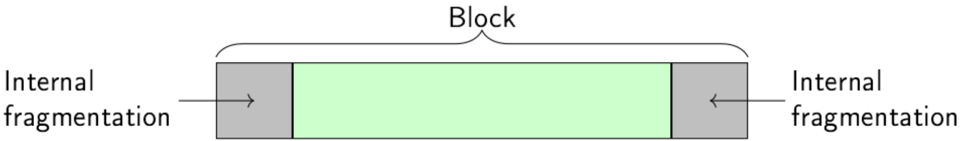
- $U_k = \max_{i<k} \frac{P_i}{H_k}$
- Indicates how much memory we have *reserved* without *using*.

Fragmentation

- **Poor memory utilisation is caused by**
  - *internal* fragmentation, and
  - *external* fragmentation.

Internal fragmentation

***Internal fragmentation* occurs when the payload is smaller than the block size.**



Causes

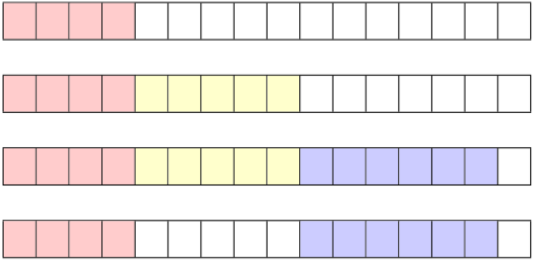
- Overhead of maintaining heap data structures.
- Padding for alignment purposes.
- Explicit policy decisions, e.g. returning a bigger block than necessary because it's faster.

Depends only on *previous* requests, so easy to measure.

External fragmentation

Occurs when there is enough aggregate heap memory, but no single free block is large enough.

```
p1 = malloc(4*4);
p2 = malloc(5*4);
p3 = malloc(6*4);
free(p2);
```



```
p4 = malloc(6*4);
```

No free block with room for six words.

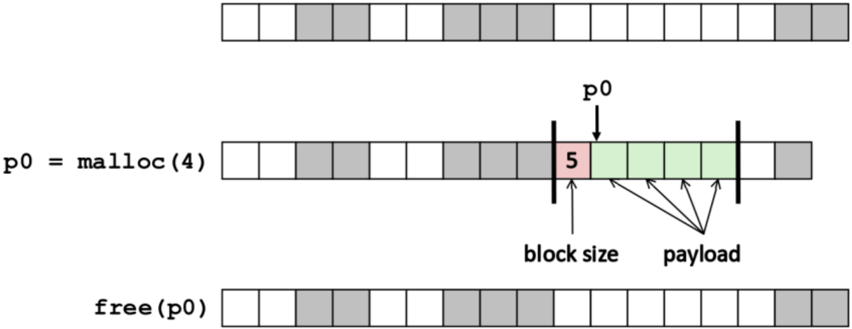
Depends on future requests, so difficult to measure.

Key implementation questions

- How does `free()` know the size of the block given just a pointer?
- How do we keep track of free blocks?

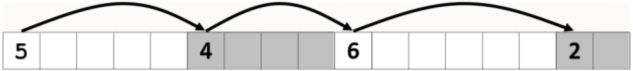
Knowing how much to free (standard method)

- Keep the length of a block in the word preceding the block.
  - This word is often called the *header field* or *header*.
  - We can also use more words to store even more metadata if we wish.
- Requires an extra word for every allocated block.

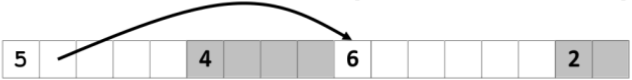


Keeping track of free blocks

- Method 1: *Implicit list* using length—links all blocks.



- Method 2: *Explicit list* among the free blocks, using pointers or offsets.



- Method 3: *Segregated free list*, different free lists for different sizes of blocks.

Conclusions

- `mmap()` syscall allows processes to map virtual memory.
  - Can map files to memory, or make anonymous mapping.
  - Can share memory between processes.
- `malloc()` is a *userspace* memory manager.
  - Not a system call itself.
  - Requests memory from kernel with `mmap()` and `sbrk()` and then parcels it out.
  - *Internal fragmentation* is when allocated blocks have wasted space.
  - *External fragmentation* is when free space is split into many small blocks.