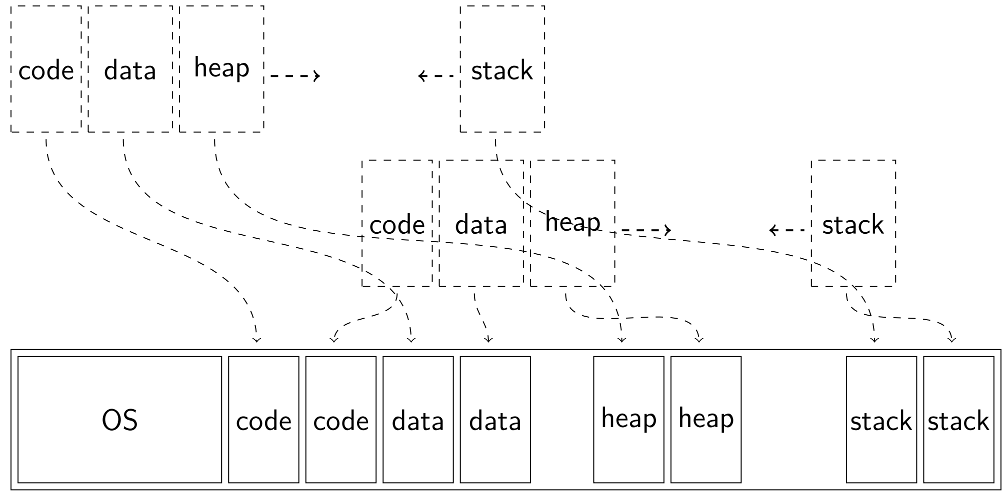
Memory management

Johan Montelius

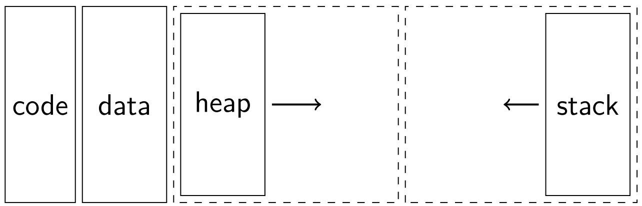
KTH

2020

virtual memory and segmentation



the process view



How do we obtain more memory for the heap data structures?

Linux system call

```
#include <unistd.h>

int brk(void *addr);
void *sbrk(intptr_t incr);
```

brk() and sbrk() change the location of the program break, which defines the end of the process's heap segment

brk() sets the end of the heap segment to the value specified by addr

sbrk() increments the program's heap space by increment bytes. It returns the previous program break.

Calling sbrk() with an increment of 0 can be used to find the current location of the program break.

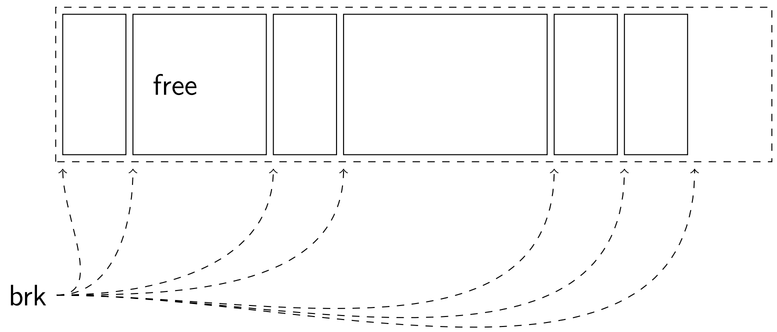
C program - not the way to do it

```
#include <stdlib.h>
#include <unistd.h>

int *allocate_array_please(int size) {

  return (int*)sbrk(size * sizeof(int));
}
```


a growing heap



How do we reuse allocated memory?

C program

```
#include <stdlib.h>

int global = 42;

int main(int argc, char *argv[]) {

  if(argc < 2) return -1;

  int n = atoi(argv[1]);

  int on_stack[5] = {1,2,3,4,5};

  int *on_heap = malloc(sizeof(int)*n);

    :
```


The POSIX API

```
#include <stdlib.h>

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

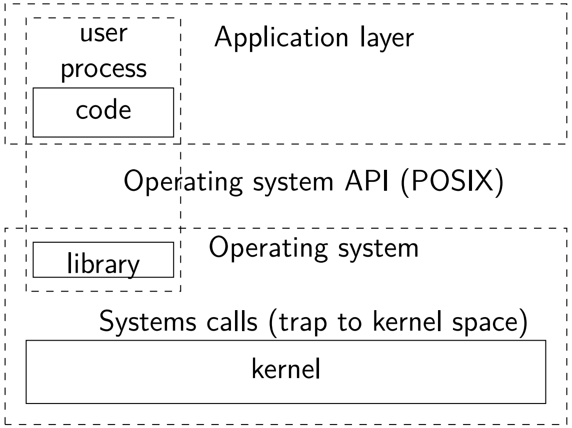
The malloc() function allocates size bytes and returns a pointer to the allocated memory. The memory is not initialized.

If size is 0, then malloc() returns either NULL, or a unique pointer value that can later be successfully passed to free().

The free() function frees the memory space pointed to by ptr, which must have been returned by a previous call to malloc(), ..

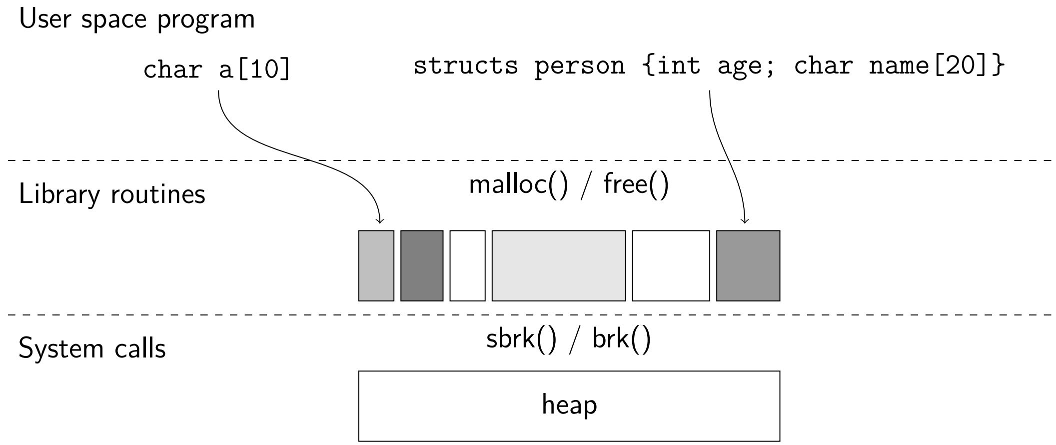
:

The operating system



Library is often just a wrapper for the system call - sometimes more complex.

Memory hierarchy



Memory management

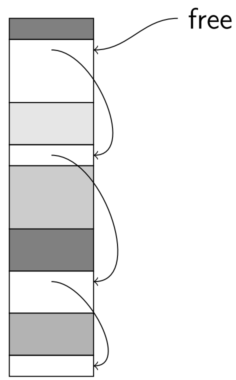
If we would not have to reuse freed memory areas - management would be simple.

- Calling `sbrk()` is costly i.e. better to do a few large allocations and then do several smaller `malloc()` operations.
- Keep track of freed memory, to reuse it in following `malloc()`.

A list of free blocks

Assume each free block holds a header containing: the size and a pointer to the next block.

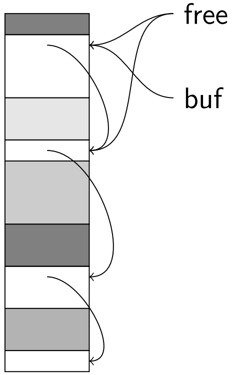
```
typefdef struct __node_t {
  int    size;
  struct __node_t *next;
}
```



pick a suitable block

When we malloc we first search the free-list for a suitable block.

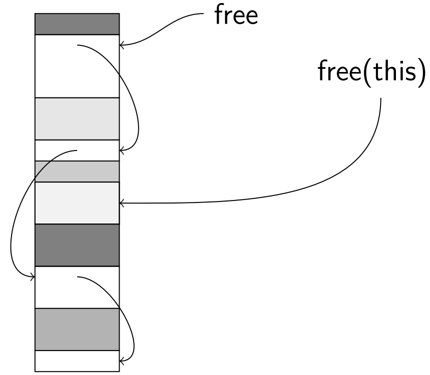
```
char *buf = malloc(20)
```



return a block

How do we return a block?

```
free(this)
```

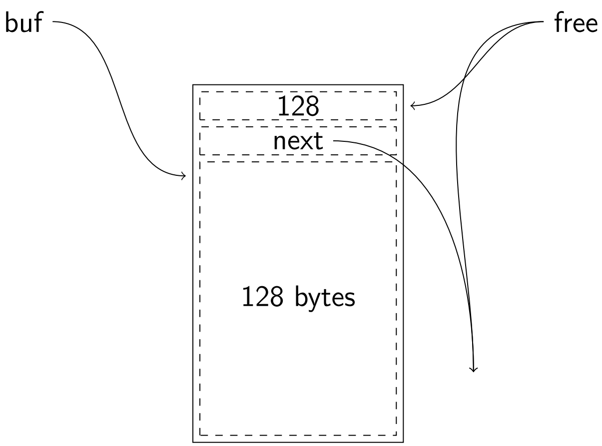


What's the problem?

hidden information

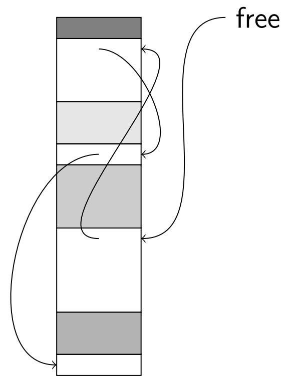
```
 :
char *buf = malloc(128);
 :

 free(buf);
 :
```



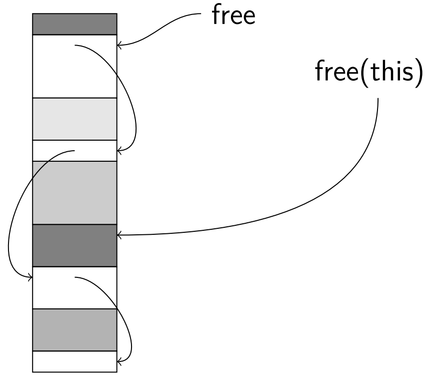
Malloc - find a suitable block and split it.

Which block shall we pick?



Coalescing - merging free blocks

When we return a block we need to merge it with adjacent free blocks - if any.



Free list strategies

- **Best fit:** the block that minimize the left over.
- **Worst fit:** the block that maximize the left over.
- **First fit:** pick the first one.

You should know the pros and cons of these strategies.

Segregated lists

Idéa - keep separate lists of blocks of different size.

Assume we keep lists for blocks of: 8, 16, 32, 64 ... bytes.

- Easy to serve and return blocks of given size.
- What should we do if we are asked for block of size 24?
- What sizes should we choose, what needs to be considered?

We can build our own allocator that is optimized for a given application.

malloc in GNU/Linux

The C standard library glibc used in most GNU/Linux distributions use a memory allocator called `ptmalloc3` (pthread malloc).

Multithreaded, each thread has a separate heap.

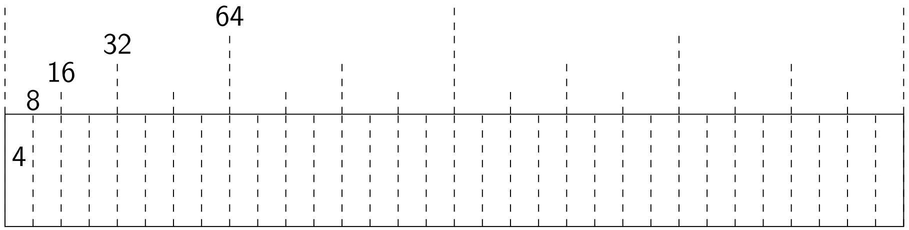
Uses multiple *bins* (free lists) to keep *chunks* of different size.

Will coalesce adjacent chunks.

Buddy Allocation

If we should allow blocks to be divided then we should also provide efficient coalescing.

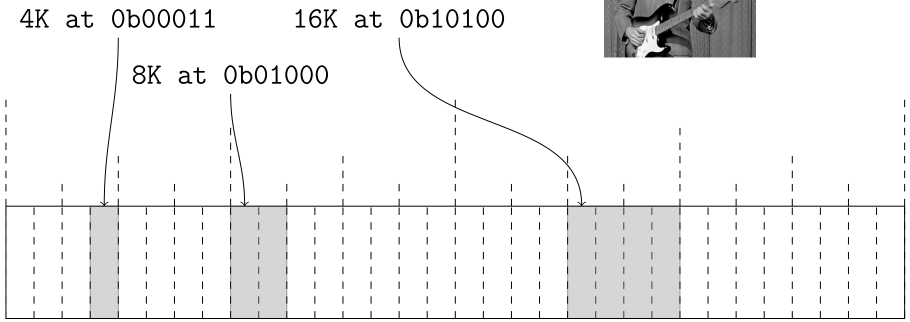
Assume total memory 128Kibyte, smallest allocated *frame* 4Kibyte



Find your buddy

Assume we number our 32 frames from 0b00000 to 0b11111.

Who's the buddy of:



Buddy pros and cons

Pros:

- Efficient allocation and deallocations of frames.
- Coalescing efficient, $O(lg(n))$
- Handles external fragmentation well.

Cons:

- Internal fragmentation - if we need a frame of 9 blocks we get 16!

Linux uses Buddy allocations when managing physical memory - check `/proc/buddyinfo`.

mmap - memory map

```
#include <sys/mman.h>

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

mmap() creates a new mapping in the virtual address space of the calling process.

The length argument specifies the length of the mapping.

If addr is NULL, then the kernel chooses the address at which to create the mapping;

The prot argument describes the desired memory protection of the mapping..

flags, fd and offset for mapping of file in memory

Originally from 4.2BSD, default in OSX where malloc() uses mmap() to allocate memory.

sbrk() vs mmap()

brk() and sbrk()

- easy to extend the process heap
- not easy to hand back allocated memory
- only one “heap”
- not part of POSIX

mmap()

- POSIX standard
- easy to allocate several large areas
- easy to hand back allocated memory
- ability to map a file in memory

Programmers point of view

- Explicit memory management: the programmer needs to explicitly free objects.
  - Used in C, C++ and most system level programming languages.
  - Pros: efficient usage of memory.
  - Cons: hard to find bugs when you don’t do it right.
- Implicit memory management: memory is freed by the system.
  - Managed by the runtime system i.e. a garbage collector (Java, Erlang, Python, ..) or by the compiler (Mercury, Rust ...).
  - Pros: much simpler and/or safer.
  - Cons: could result in runtime overhead and/or lack of control.

Summary

- user process API: malloc() and free()
- system calls: sbrk() or mmap()
- how to find suitable memory block.
- how to free memory blocks for efficient reuse
- coalescing smaller blocks