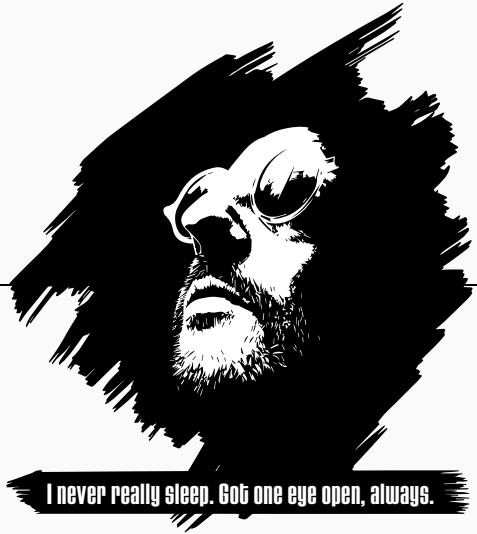
Malloc - Presentation

ACU 2020 Team



Copyright

Rules

- You must have downloaded your copy from the Assistants' Intranet <https://intra.assistants.epita.fr>.
- This document is strictly personal and must **not** be passed on to someone else.
- Non-compliance with these rules can lead to severe sanctions.

Introduction

A user/system interface

- `malloc` is a *wrapper* around system-provided facilities for memory management.
- It reserves memory from the system then uses various algorithms to allocate blocks for the user within that memory.

What you will learn

Memory allocation functions

- `malloc`
- `free`
- `realloc`
- `calloc`

Memory management

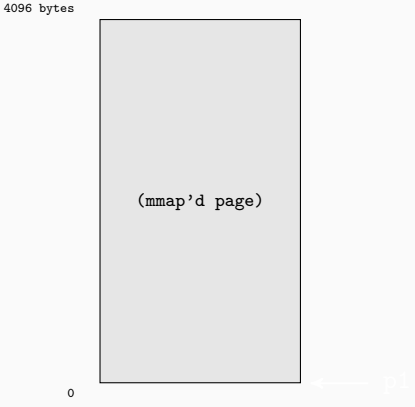
- `sbrk`
- `mmap`

Allocation strategy

- First-fit
- Best-fit
- Binary buddies
- ...

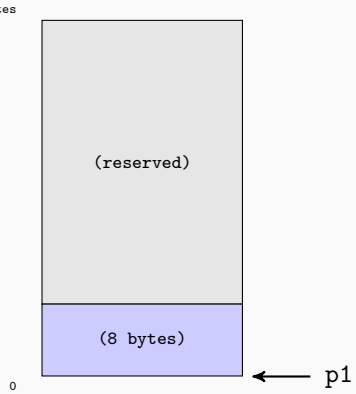
Malloc 101

Example



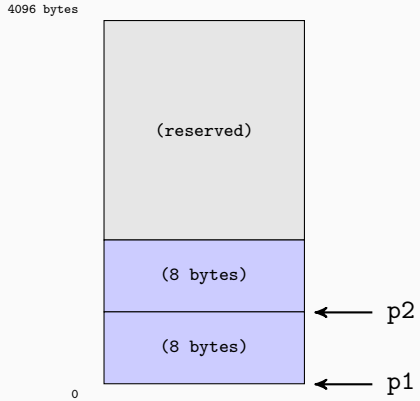
Example

```
void *p1 = malloc(8);
```



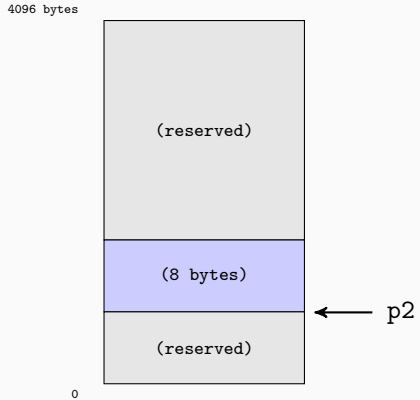
Example

```
void *p1 = malloc(8);
void *p2 = malloc(8);
```



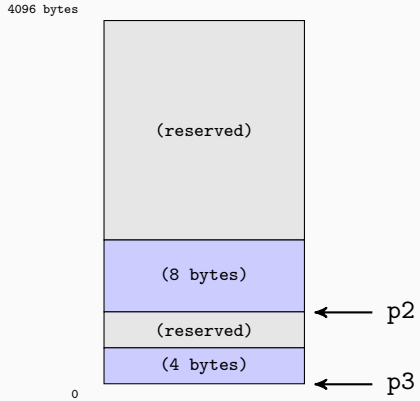
Example

```
void *p1 = malloc(8);
void *p2 = malloc(8);
free(p1);
```



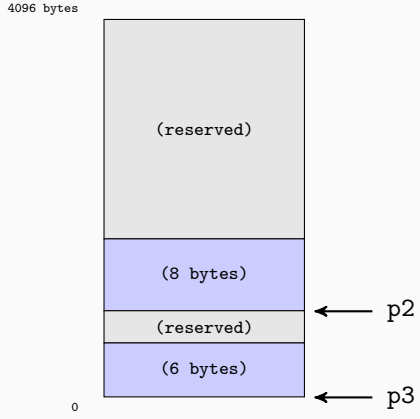
Example

```
void *p1 = malloc(8);
void *p2 = malloc(8);
free(p1);
void *p3 = malloc(4);
```



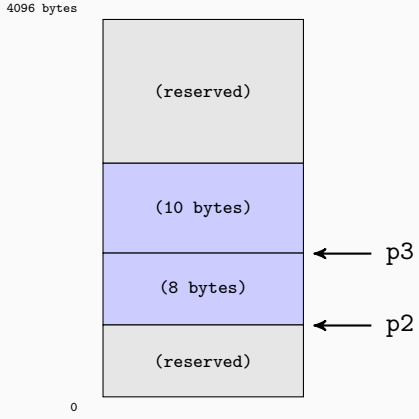
Example

```
void *p1 = malloc(8);
void *p2 = malloc(8);
free(p1);
void *p3 = malloc(4);
p3 = realloc(p3, 6);
```



Example

```
void *p1 = malloc(8);
void *p2 = malloc(8);
free(p1);
void *p3 = malloc(4);
p3 = realloc(p3, 6);
p3 = realloc(p3, 10);
```



System-side memory management

Dynamic memory allocation

Two ways of getting memory from the system:

- `sbrk(2)`
- `mmap(2)`

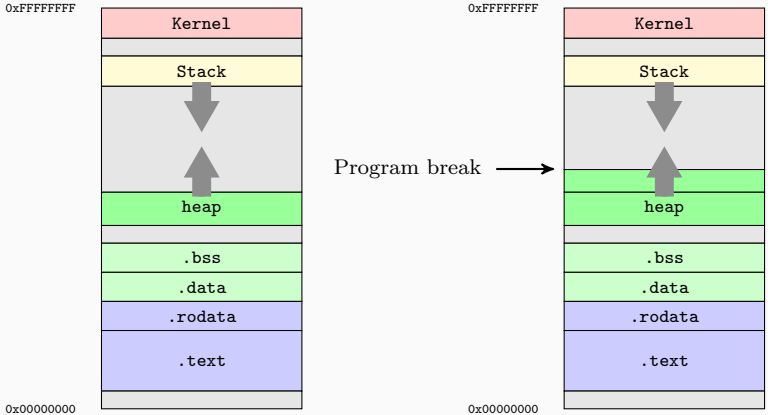
sbrk (2)

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

Change the data segment size.

- Interface created to match *segmented memory management* systems available on most CPUs.
- Legacy interface, we will not use it.

Heap example



mmap (2)

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

Map pages to a virtual address

- `addr`: starting address (hint)
- `length`: number of bytes to map
- `prot`: permissions (`PROT_READ`, `PROT_WRITE`, ...)
- `flags`: options (`MAP_PRIVATE`, `MAP_ANONYMOUS`, ...)
- `fd`: descriptor of file to be mapped; -1 with `MAP_ANONYMOUS`
- `offset`: starting offset in the mapped file

Don't forget to define the right feature test macros to be able to use all the flags. For more information, see `man 2 mmap`.

Be careful, you **MUST** check mmap return value (see MAP_FAILED macro defined in `sys/mman.h`).

Memory Mapping

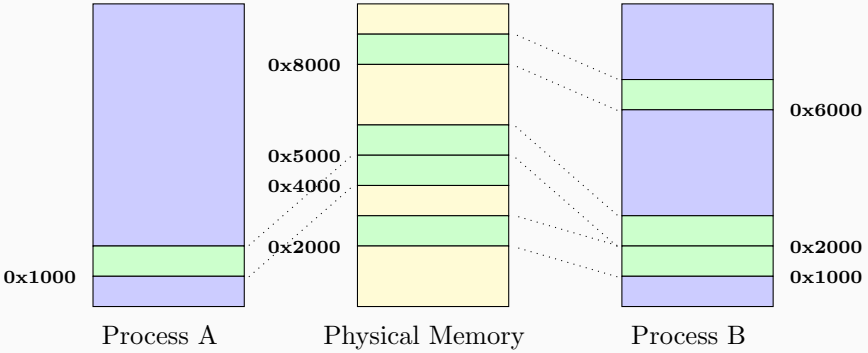
- You can use `mmap` (2) to map plain memory, thus reserving memory for the process. This is called an *anonymous mapping*.
- `mmap`'ed memory can be released with `munmap` (2) and resized using `mremap` (2)
- Lots of *syscalls* related to memory management: see `sys/mman.h`.
- Memory mapping is backed by paging.

Example

my_get_page.c

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

void *my_get_page(void)
{
    void *addr = mmap(NULL, 4096, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    if (addr == MAP_FAILED)
        return NULL;
    return addr;
}
```


0x8000
0x5000
0x4000
0x2000
0x1000
0x6000
0x2000
0x1000
Process A
Physical Memory
Process B

Algorithms

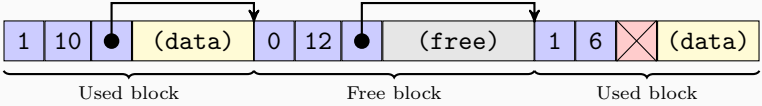
Metadata

- `malloc` returns blocks matching the required size.
- `mmap` allocates an empty memory zone.

$\Rightarrow$ To manage these memory zones, **metadata is needed**.

Metadata usually includes:

- block state
- block size
- pointer to the next block



Strategies

- Several allocation strategies, each with its own features.
- What matters:
  - performance
  - efficiency (little fragmentation)
  - portability

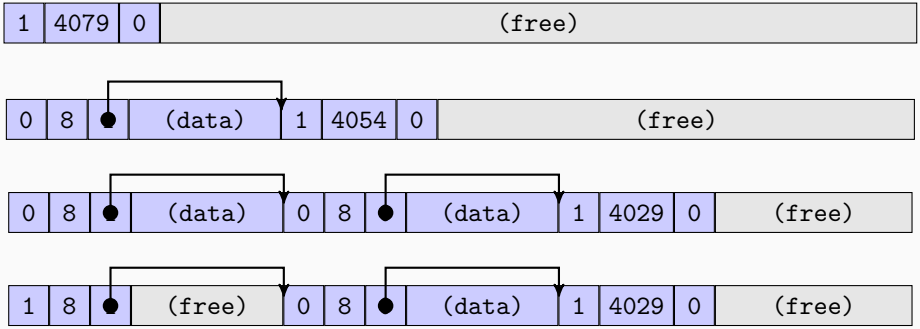
Cheating

Doing `malloc = mmap` is not an algorithm and will be considered cheating!

First-fit

- Look for the first block in the list satisfying the size requirement.
- If the block is larger than necessary, the unused space is put back in a new block.
- Heavy fragmentation, accumulation of small blocks.
- Still an effective technique.
- Variants: best-fit, worst-fit...

Schema first-fit



Size buckets

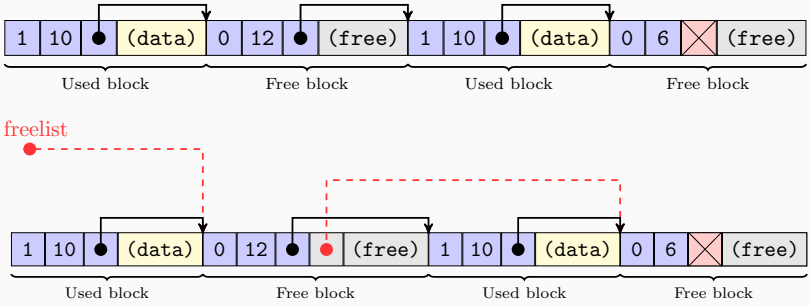
0010101100
(16 bytes)
(16 bytes)
(16 bytes)
(16 bytes)
(16 bytes)
(16 bytes)
(16 bytes)
(16 bytes)
(16 bytes)
(16 bytes)

1100010100
(32 bytes)
(32 bytes)
(32 bytes)
(32 bytes)
(32 bytes)
(32 bytes)
(32 bytes)
(32 bytes)
(32 bytes)
(32 bytes)

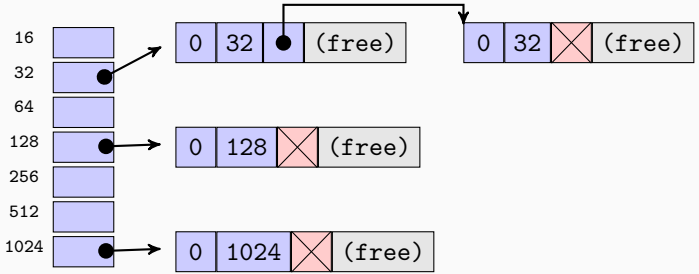
0110001101
(8 bytes)
(8 bytes)
(8 bytes)
(8 bytes)
(8 bytes)
(8 bytes)
(8 bytes)
(8 bytes)
(8 bytes)
(8 bytes)

Improvements

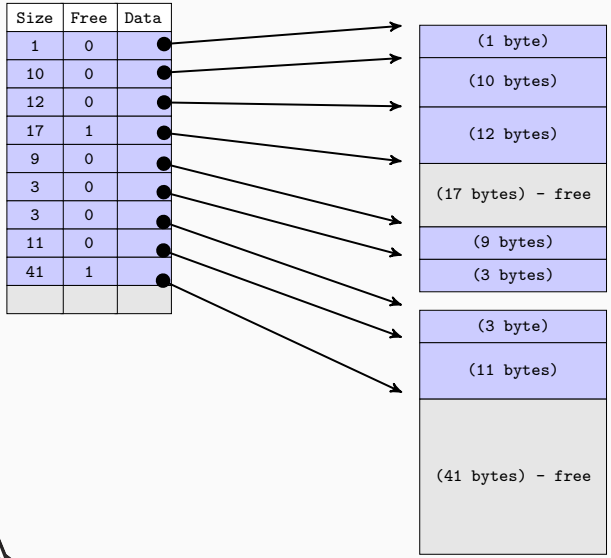
Free-list


1
10
(data)
0
12
(free)
1
10
(data)
0
6
(free)
Used block
Free block
Used block
Free block
freelist

Sized free-lists



External metadata



Survival kit

Using your allocator

Your memory allocation functions should be exported in a `libmalloc.so` shared object.

How to use your malloc:

- Link against `libmalloc.so` and use `LD_LIBRARY_PATH`:

```
42sh$ gcc main.c -L. -lmalloc -o my_program
42sh$ LD_LIBRARY_PATH=. ./my_program
```

- Just use the `LD_PRELOAD` environment variable:

```
42sh$ LD_PRELOAD=./libmalloc.so ls
```

Testing your allocator

- We will use `LD_PRELOAD` to run several existing programs with your allocator.
- Be sure to test it thoroughly by doing the same.

We want to see you use a test suite. A shell script is acceptable and is enough for this project. But we want multiple and different tests.

Debugging your allocator

- You only want the debugged program to use your library and not gdb especially when your library is buggy. That's why you can't launch gdb as shown previously.
- You have to start gdb normally and then set the debugged program environment using gdb commands to ensure that it will use your library.

```
42sh$ ls
include  libmalloc.so  main  main.c  main.o  Makefile  malloc.c
42sg$ gdb ./main
Reading symbols from ./main...done.
(gdb) set env LD_LIBRARY_PATH=.
(gdb) start
```

Note: Don't forget to compile your binary and your library with `-g`.

Debugging other programs with your library preloaded

```
42Sh$ gdb /bin/ls
...
Reading symbols from /bin/ls...(no debugging symbols found)...done.
(gdb) set exec-wrapper env 'LD_PRELOAD=./libmalloc.so'
(gdb) break malloc.c:malloc
No symbol table is loaded.  Use the "file" command.
Make breakpoint pending on future shared library load?

Breakpoint 1 (malloc.c:malloc) pending.
(gdb) run
Starting program: /bin/ls

Breakpoint 1, malloc (size=568) at malloc.c:307
307 {
(gdb)
```

Tracing function calls

It may be useful to display a message every time a function from your `libmalloc.so` is called.

A file `call_trace.c` is provided in order to help you.

`call_trace.c` should be compiled as:

```
42sh$ gcc call_trace.c -g -shared -fPIC -o libtracemalloc.so -ldl
```

Tracing the default allocator

The following line:

```
42sh$ LD_PRELOAD=./libtracemalloc.so ls
```

will use the function wrappers contained in `call_trace.c` and print to stderr before entering and after exiting from `malloc`, `free`, `calloc` and `realloc`. This is done using the default functions.

Tracing your allocator

In order to use your `libmalloc.so`, you should use the following line:

```
42sh$ LD_PRELOAD=./libtracemalloc.so:./libmalloc.so ls
```

You are now able to compare the calls done with your `libmalloc.so` and the calls done by the default functions.

Example tracing an allocator (code)

This allocator does literally nothing.

```
42sh$ cat malloc.c
#include <stddef.h>

void *malloc(size_t size)
{
    return NULL;
}

void free(void *ptr)
{
    return;
}
```

Example tracing a bad allocator

```
42sh$ gcc -Wall -Wextra -Werror -pedantic -shared -fPIC -o libmalloc.so malloc.c
42sh$ LD_PRELOAD=./libtracemalloc.so:./libmalloc.so factor 10
[!] entering malloc(5)
[!] exiting malloc(5) = (nil)
[!] entering malloc(552)
[!] exiting malloc(552) = (nil)
[!] entering malloc(5)
[!] exiting malloc(5) = (nil)
[!] entering free((nil))
[!] exiting free((nil))
[...]
[!] entering malloc(1024)
[!] exiting malloc(1024) = (nil)
[!] entering free((nil))
[!] exiting free((nil))
factor: memory exhausted
```

Example tracing the default allocator

```
42sh$ LD_PRELOAD=./libtracemalloc.so factor 10
[!] entering malloc(5)
[!] exiting malloc(5) = 0x562b0bcbe260
[!] entering free(0x562b0bcbe260)
[!] exiting free(0x562b0bcbe260)
[!] entering malloc(120)
[!] exiting malloc(120) = 0x562b0bcbe280
[!] entering malloc(12)
[!] exiting malloc(12) = 0x562b0bcbf270
[...]
[!] entering malloc(10)
[!] exiting malloc(10) = 0x562b0bcc04a0
[!] entering malloc(1024)
[!] exiting malloc(1024) = 0x562b0bcc04c0
10: 2 5
```

Miscellaneous advice

- Start **early**.
- Read the **entire** subject *AND* the man pages.
- Don’t spend too much time on complicated algorithms.

Newsgroup `assistants.projets`, `[MLL]` tag.
Deadline October 20, 11:42

As usual:

- Your project must comply with the coding style.
- Cheating will be sanctioned.

Questions

Any questions?