Memory Management (Part 3)

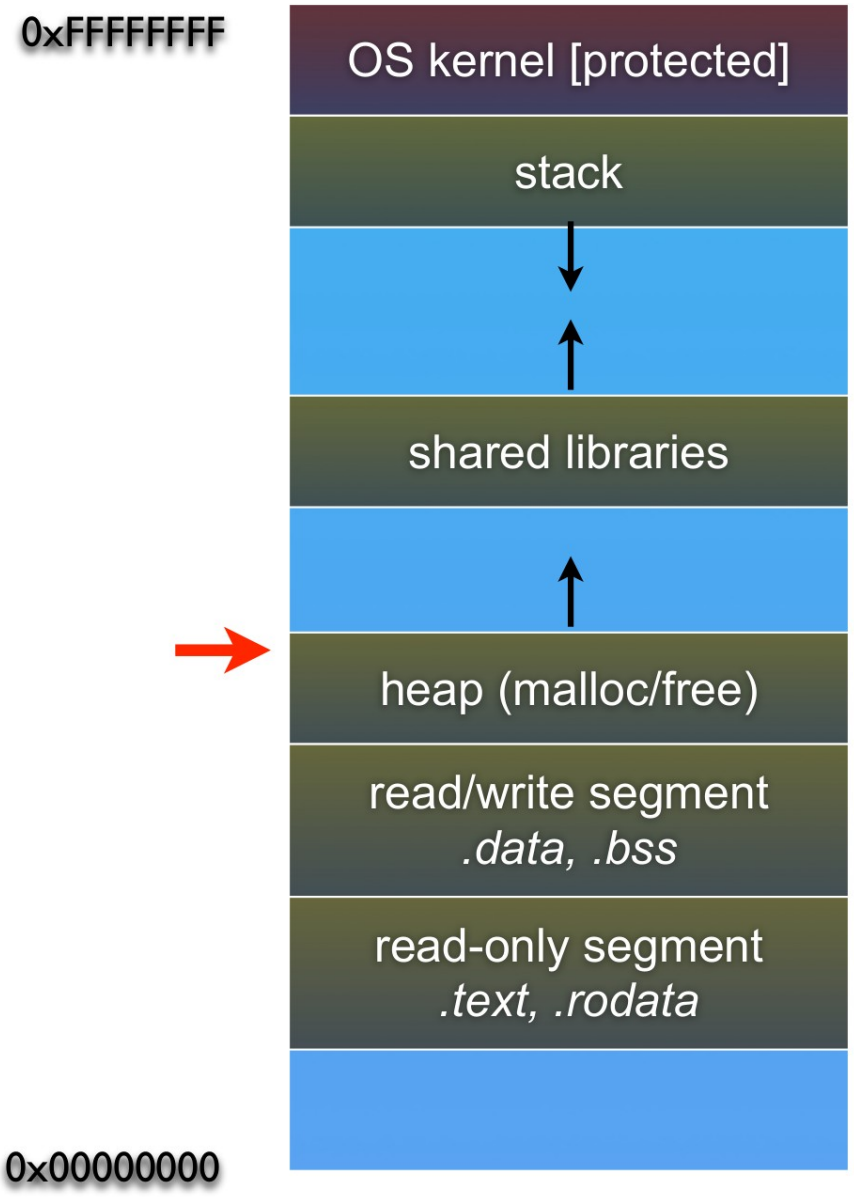
Devin J. Pohly <djpohly@cse.psu.edu>

Whence virtual memory?

- Every process begins with a certain amount of memory reserved for the heap
  - The top of the heap is known as the *program break*
  - Functions like `malloc` and `free` manage the heap by obtaining and releasing memory
    - You don't see it because it's handled for you

The program break

- The program break is moved up and down by `malloc` and `free`
  - Moved up to reserve more memory for the heap
  - Moved down to release some memory previously reserved for the heap



brk and sbrk

- These functions are used to manage the program break
  - `int brk(void *addr)`: moves the program break to be at the address in `addr` (!)
    - This is an absolute address, so you can really mess things up – for example, by passing a pointer that points inside the stack.
  - `void *sbrk(long inc)`: moves the program break up by `inc` bytes
    - Positive `inc` increases the space reserved for the heap
    - Negative `inc` reduces the space reserved for the heap
    - Calling `sbrk(0)` will return the current program break without changing it

Let's try it

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

void check_memory() {
        static void *last = NULL;

        void *ptr = sbrk(0);
        if (last)
                printf("The top of the heap is [%p] (%+d)\n", ptr, ptr - last);
        else
                printf("The top of the heap is [%p]\n", ptr);
        last = ptr;
}

int main()
{
        void *xptr[2048];
        int i;

        check_memory();
        xptr[0] = malloc(0x1000);
        check_memory();
        for (i = 1; i < 1024; i++) {
                xptr[i] = malloc(0x1000);
        }
        check_memory();
        for (i = 0; i < 1024; i++) {
                free(xptr[i]);
        }
        check_memory();
        return 0;
}
```

Let's try it

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

void check_memory() {
        static void *last = NULL;

        void *ptr = sbrk(0);
        if (last)
                printf("The top of the heap is [%p] (%+d)\n", ptr, ptr - last);
        else
                printf("The top of the heap is [%p]\n", ptr);
        last = ptr;
}

int main()
{
        void *xptr[2048];
        int i;

        check_memory();
        xptr[0] = malloc(0x1000);
        check_memory();
        for (i = 1; i < 1024; i++) {
                xptr[i] = malloc(0x1000);
        }
        check_memory();
        for (i = 0; i < 1024; i++) {
                free(xptr[i]);
        }
        check_memory();
        return 0;
}
```

```
$ ./brktest
The top of the heap is [0x1b78000]
The top of the heap is [0x1b9a000] (+139264)
The top of the heap is [0x1f99000] (+4190208)
The top of the heap is [0x1b99000] (-4194304)
```

Some observations

- The program (via `malloc`) kept getting more and more memory over time.
- It released some, but not all of memory at the end.
  - Some "slack" is left over.

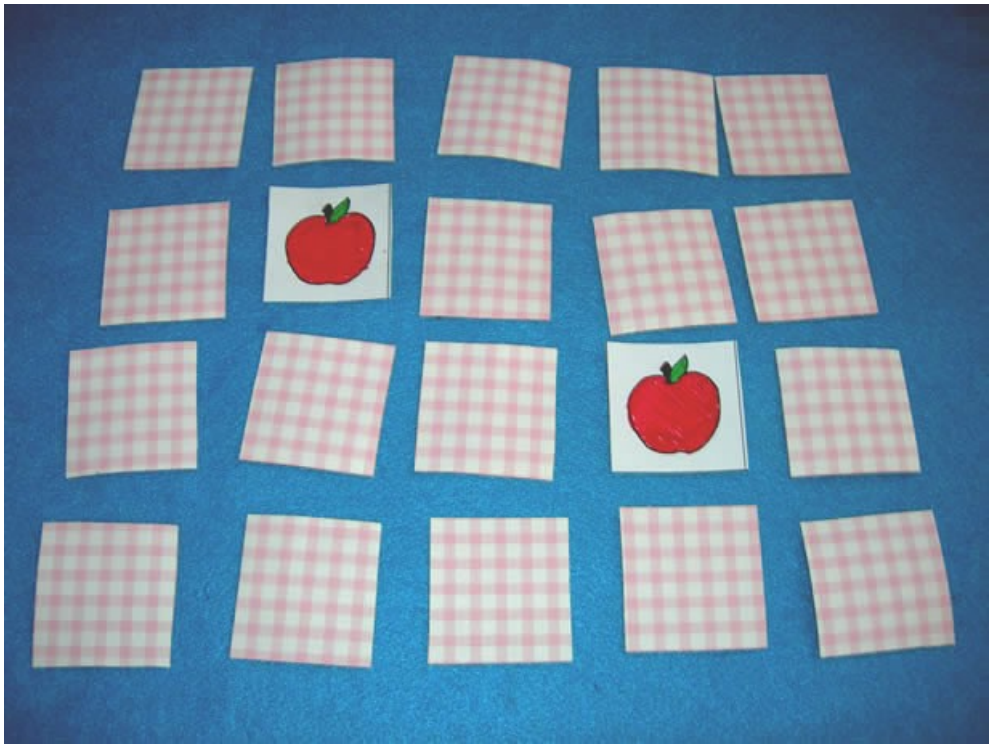
Top of heap	Change
0x1b78000	
0x1b9a000	+0x0022000
0x1f99000	+0x03ff000
0x1b99000	-0x0400000

An alternate method

- Some implementations of `malloc` don't use the program break; instead, they "memory map" a region to reserve more memory
  - In essence, this tells the OS that a certain range of memory should be available for use by the program.
  - This can be anywhere in the process address space (i.e., regardless of the program break).

Memory mapping

- In its most general use, the memory map system call "maps" (overlays) a file onto physical memory.
  - A program can then treat the file as random-access memory.
  - Very fast, because the OS "pages" the file content into memory in chunks (often 4kiB).
  - Note: memory mapping is often used to load program code when executing



Mapping memory

- To map memory, use the `mmap` function:

```
void *mmap(addr, len, prot, flags, fd, ofs);
```

  - `void *addr`: address at which the file should be mapped
    - If this is NULL, the OS will choose where to map it.
  - `size_t len`: how much memory to map
  - `int prot`: protection bits (permissions)
  - `int flags`: bits specifying the type of mapping
  - `int fd`: file descriptor of the file to map (see I/O lectures)
  - `off_t ofs`: starting offset within the file
- Returns a pointer to the region or "`(void *) -1`" on error

mmap protection bits

- The `prot` argument specifies the desired memory protection of the mapping.
  - It is either `PROT_NONE` or the bitwise OR of one or more of the following flags:
    - `PROT_READ`: memory may be read
    - `PROT_WRITE`: memory may be written
    - `PROT_EXEC`: memory may be executed
- These bits allow the program to police the use of mapped files/memory regions.

mmap flags

- Flags indicate what kind of mapping we are doing:
  - `MAP_SHARED`: ensures that updates to the mapping are written to the underlying file and visible to other processes that map it
  - `MAP_PRIVATE`: creates a private, copy-on-write mapping in which updates are not written to the underlying file or visible to other processes
- And one other option:
  - `MAP_ANONYMOUS`: the mapping is not backed by any file, and its contents are initialized to zero

Unmapping memory

- The complement to `mmap` is `munmap`:

```
void *munmap(addr, len);
```

  - `void *addr`: address of the memory to be released
    - Must be memory that was originally mapped with `mmap`!
  - `size_t len`: how much memory to release

- Note: any remaining changes to a `MAP_SHARED` mapping will be written to disk at this point

Mapping example

```c
int mymap(char val) {
        // Local variables
        int fd, i;
        char *ptr;

        // Open the file and map 20 bytes to memory
        fd = open("mmap.dat", O_CREAT | O_RDWR);
        if (fd < 0) {
                perror("open");
                return 1;
        }

        ptr = mmap(NULL, 20, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
        if (ptr == (void *) -1) {
                perror("mmap");
                return 1;
        }

        // Add the values
        for (i = 0; i < 20; i++) {
                ptr[i] = val;
        }

        // Release the mapped memory and close the file
        munmap(ptr, 20);
        close(fd);
        return 0;
}
```

Mapping example

```c
int mymap(char val) {
        // Local variables
        int fd, i;
        char *ptr;

        // Open the file and map 20 bytes to memory
        fd = open("mmap.dat", O_CREAT | O_RDWR);
        if (fd < 0) {
                perror("open");
                return 1;
        }

        ptr = mmap(NULL, 20, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
        if (ptr == (void *) -1) {
                perror("mmap");
                return 1;
```

```
(gdb) x/20xb ptr
0x7ffff7ff6000: 0x7f    0x45    0x4c    0x46    0x02    0x01    0x01    0x00
0x7ffff7ff6008: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7ffff7ff6010: 0x02    0x00    0x3e    0x00

(gdb) next
...

(gdb) x/20xb ptr
0x7ffff7ff6000: 0x58    0x58    0x58    0x58    0x58    0x58    0x58    0x58
0x7ffff7ff6008: 0x58    0x58    0x58    0x58    0x58    0x58    0x58    0x58
0x7ffff7ff6010: 0x58    0x58    0x58    0x58
```

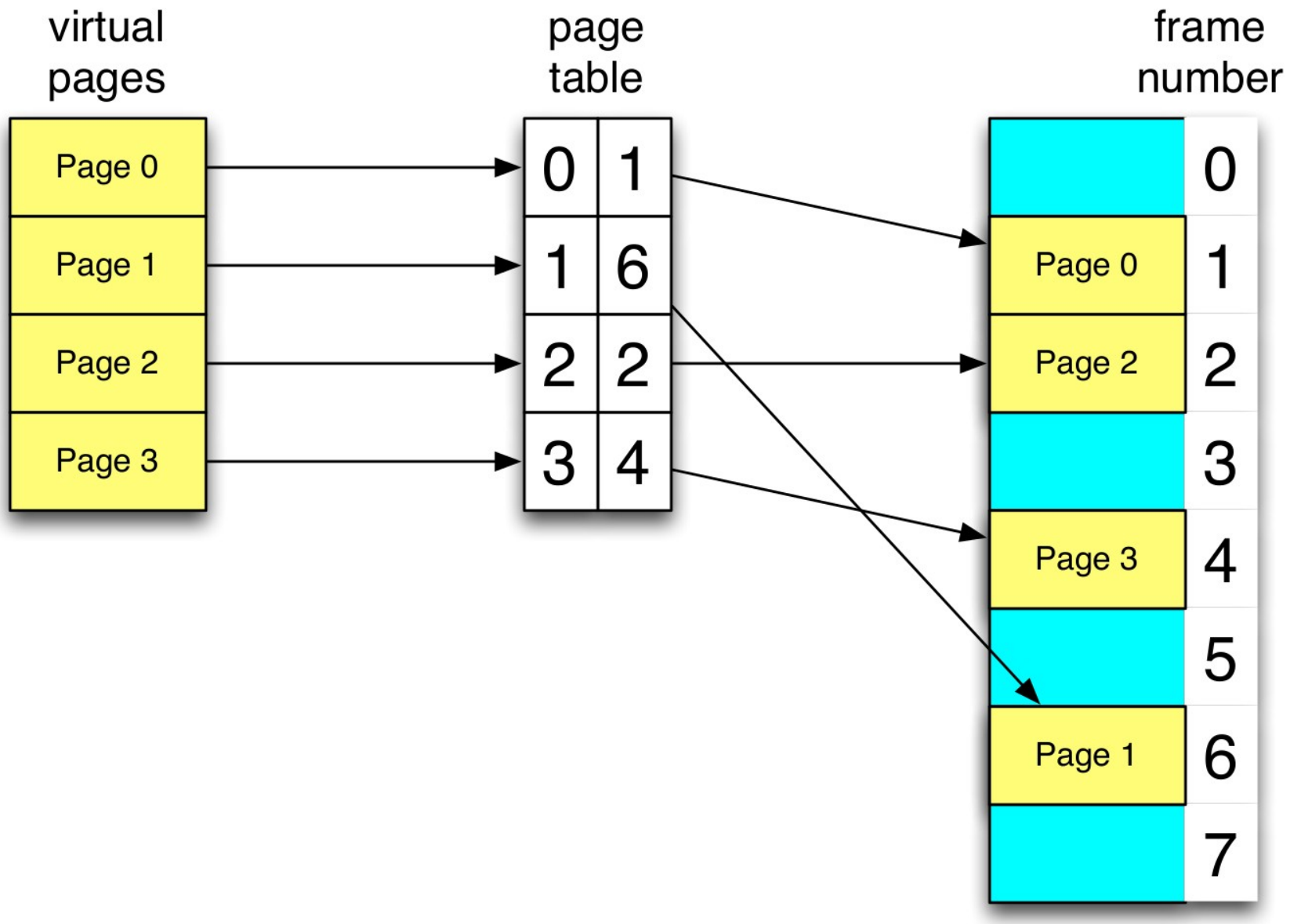
An alternate method

- So an implementation of `malloc` and `free` can use:
  - `mmap` instead of `brk`/`sbrk`
  - `munmap` instead of `brk`/`sbrk`
- This is a design choice made when implementing the standard library
  - e.g., the GNU C library (glibc)

OS paging

- Programs have a virtual address space (say, 1MiB)
- OS must fetch data from either physical memory or disk
  - Done by a mechanism called *paging* (or *demand paging*)
  - Divide the virtual address space into units called *pages*, each of which is a fixed size (often 4kiB)
    - For example 1MiB virtual address space has 256 4kiB pages.
  - Physical memory is divided into *frames*, usually the same size as pages
    - For example, we may have only 32 4kiB frames.
- OS tracks where pages are stored in physical memory
  - Maintained in a data structure called the *page table*
  - Maps virtual addresses to physical addresses

Example page table



Multiple processes