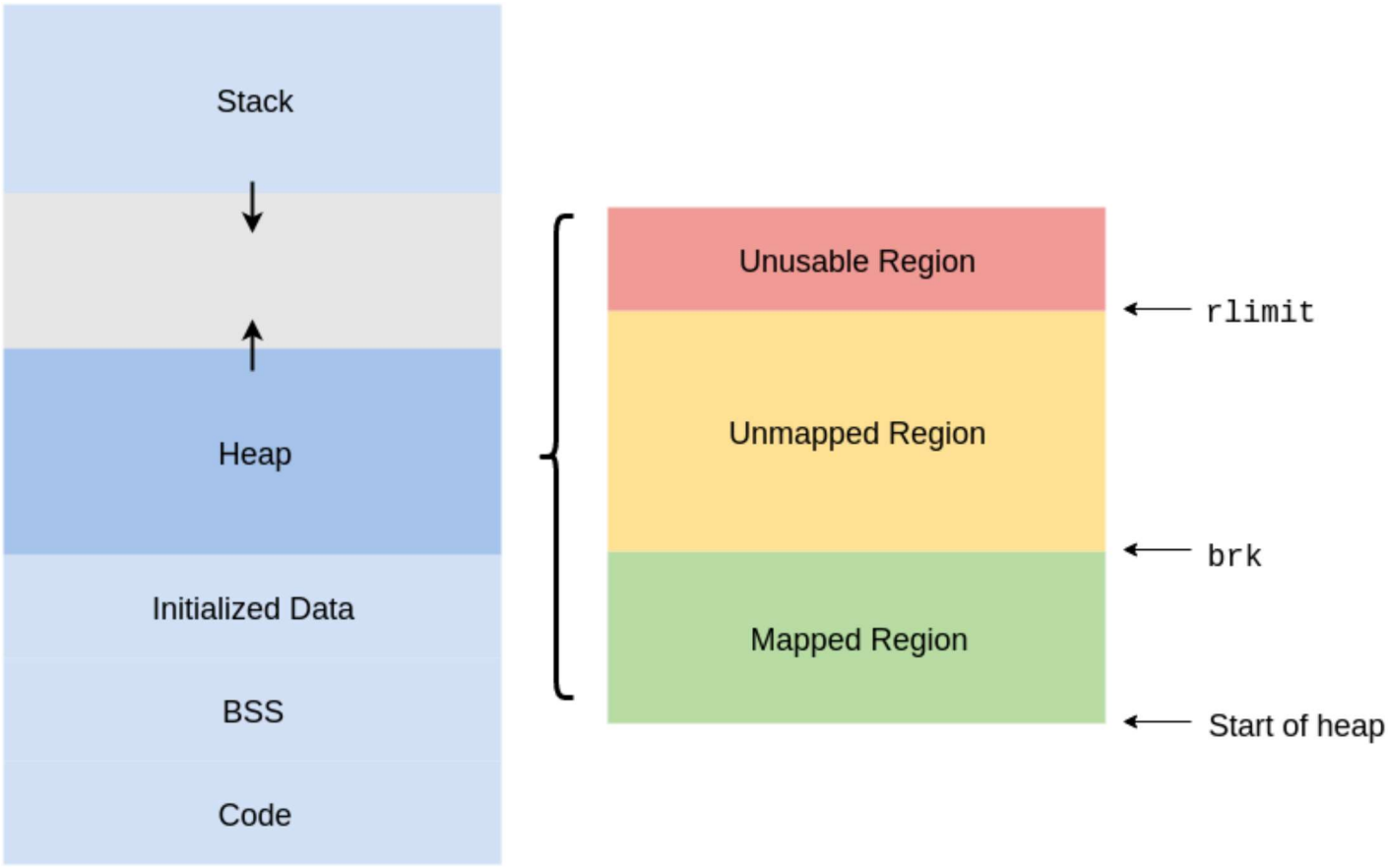
Syscall implementation

TABLE OF CONTENTS

You are advised to begin by reading over the functions `start_process` and `load` in `process.c`. Consider what new initialization you must perform in these locations in order to support the heap functionality outlined in previous sections.

Process memory

Each process has its own virtual address space. Parts of this address space are mapped to physical memory through address translation. In order to build a memory allocator, we need to understand how the heap in particular is structured. Here we describe the memory layout of a process, focusing on the structure of the heap, within a *Linux* process. You'll have to implement a simplified version of the heap in Pintos with the `sbrk` system call.



The heap is a space of memory, continuous in the virtual address space of a process, with three bounds:

- The bottom of the heap.
- The top of the heap, known as the break. The break can be changed using `brk` and `sbrk`. The break marks the end of the mapped memory space. Above the break lies virtual addresses which have not been mapped to physical addresses by the operating system. **For simplicity, you only need to implement `sbrk` in Pintos for this assignment. You do not need to bother with `brk`. You should implement `sbrk` as a new system call in Pintos.**
- The hard limit of the heap, which the break cannot surpass. See Resource limits for more information. **For simplicity, you should not implement a hard limit on the heap size for this assignment.**

In this assignment, you'll be allocating blocks of memory in the mapped region and moving the break appropriately whenever you need to expand the mapped region.

Requesting memory from the operating system

Initially the mapped region of the heap will have a size of 0. To expand the mapped region, we have to manipulate the position of the break. The way to do this is via `sbrk`, defined in `lib/user/syscall.c`:

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

Your task is to implement the `sbrk` syscall by implementing a `syscall_sbrk` function inside `src/userprog/syscall.c` that will increment the position of the break by `increment` bytes and returns the address of the previous break (i.e. the beginning of newly mapped memory if `increment` is positive). You will need to add appropriate changes to `src/lib/syscall-nr.h`, `src/lib/user/syscall.c`, and the syscall handler in `src/userprog/syscall.c`. To get the current position of the break, pass in an `increment` of 0. Run `man 2 sbrk` on Linux for additional useful information.

To implement the `syscall_sbrk` function, you'll need to keep track of two variables for each process: the start of the heap, and the segment break (end of the heap). These should be maintained in the `struct thread`. We describe how to how to use these variables below.

Determining the start of the heap

You should make sure that a process' heap is located above (i.e. at a higher virtual address than) the process' code and other data loaded from the executable. You should determine at what address the heap should start when the program is loaded, after which it should remain fixed for the duration of the process. Look closely at the `load` function in `process.c`. For each loadable segment in the executable (remember segments from Homework 0?), `load` allocates pages in the virtual address space of the process, according to the read/write permissions and virtual address specified in the executable, and reads data from the executable file into those pages. Based on where the segments are loaded into memory, you should determine at what address the heap should start.

The ELF executable format guarantees that loadable segments will be listed in the executable in ascending order by virtual address space. **Thus, you should start the heap at a virtual address after the last loadable segment processed by the `load` function.** We recommend choosing a page-aligned address to start the heap.

To learn more about ELF, read `man 5 elf`.

Manipulating the segment break

The segment break should be the first address past the end of the heap, so you can initialize the data segment break to the start of the heap after loading the process. You should move it only in response to `sbrk` system calls. **The user program should be able to write data starting at the start of the heap, up to and not including the segment break.** If the user program moves the segment break to increase the size of the heap, you should allocate pages and map them into the user's virtual address space as necessary. If the user program moves the segment break to decrease the size of the heap, you should deallocate pages that no longer contain part of the heap as

necessary. You should only have to allocate or deallocate pages if the segment break crosses a page boundary.

Memory can only be mapped into a virtual address space in quanta of pages. Therefore, if the segment break is not page-aligned, it is acceptable for memory after the system break, but before the next page boundary, to be accessible by a process. See Unmapped region and no man's land for more information.

Allocating additional pages for a process' heap will fail if, for example, the user memory pool is exhausted and `palloc_get_page` fails. If `sbrk` fails, the net effect should be that `sbrk` returns `(void*) -1` and that the segment break and the process heap are unaffected. You might have to undo any operations you have done so far in this case.

Finally, real operating systems may allocate pages for `sbrk` lazily, similar to stack growth. While `sbrk` moves the segment break, pages are not allocated until the user program actually tries to access data in its heap. **For simplicity, you are not required to implement this optimization.**

Copyright © 2022 CS 162 staff.