Homework 4: Memory

Due: April 12, 2022

Contents

1 Overview

The Pintos userspace that you implemented in Project 1 is subject to two major limitations. First, the stack - which you set up properly to load a new process - is confined to a single page. If a user program's stack grows beyond 4,096 bytes, due to a long chain of function calls (e.g. due to a recursive function) and/or stack-allocated data, then the process will be terminated on its first memory access beyond the first stack page. Second, there is no way to perform any dynamic memory allocation in a user program. Presently, all memory in a user process must either have been loaded from the executable (e.g. code, globals) or must be allocated on the stack.

In this homework, you will remove both of these limitations. You will modify Pintos to dynamically extend the stack space of a process in response to memory accesses beyond the currently allocated stack. In addition, you will provide a way for a user process to explicitly request more memory from the Pintos kernel.

1.1 Getting Started

Log in to your VM and grab the skeleton code from the staff repository.

```
> cd ~/code/personal
> git pull staff master
> cd hw-memory
```

To build the code and run the tests,

```
> cd src/memory
> make check
```

1.2 Background

1.2.1 Skeleton

This homework builds on the Pintos userspace that you implemented in Project 1 (among some other functionality implemented which is mentioned below). Unlike Project 1, however, this is an individual assignment. You should NOT share code with your group members to complete this assignment! To simplify the logistics, we are providing a minimal implementation of the requisite functionality from Project 1 to complete this assignment. This implementation is built on an older version of Pintos, and is not a thorough implementation of the functionality from Project 1 — as such, you should complete this assignment by building on the userspace implementation that we provide you in the starter code, NOT using your group's implementation from Project 1. That said, the nature of this assignment's dependency on Project 1 is that, while certain features of the Pintos userspace must be working in order for this assignment to be well-formed (e.g. you have to be able to start a user process before it makes sense to extend its stack), the code you write for this assignment will not depend directly on the implementation of Project 1 features.

The starter code implements the following.

1. The one-line `do-nothing` hack needed to complete the Project Pregame.
2. The `write` system call for file descriptor 1 (`stdout`).
3. The ability to `open`, `read`, `write`, and `close` a *single* file per process.
4. Argument validation for the provided system calls using the page fault handler.
5. Elements needed for the heap growth portion of the homework in `struct thread`.
6. The implementation for malloc/calloc/realloc/free in Pintos.

Notably, the provided starter code does **not** provide the ability to pass arguments on the command line, `exec` a new process, or open multiple files. We chose to implement only the above mentioned items to minimize the amount of code you would need to look over and simplify some of the implementation (discussed below).

Importantly, the provided starter code validates arguments to system calls using the following technique. Rather than validating every byte of user-provided arguments, the code merely checks that the arguments' memory is below `PHYS_BASE`, and then reacts to page faults by terminating the running process. This is more efficient than validating the entire buffer up front, and makes implementing stack extension easier. **You should carefully read the provided code that validates system call arguments to understand how it works.**

The following are some files you may work with throughout this assignment.

`threads/palloc.c`
: Page allocator.

`threads/vaddr.h`
: Helper functions for working with virtual addresses in Pintos.

`userprog/process.c`
: Loads ELF binaries, starts processes, and switches page tables on context switch. You should be familiar with this code based on your experience in Project User Programs.

`userprog/pagedir.c`
: Manages the page tables. You probably won't need to modify this code, but you may want to call some of these functions.

`userprog/syscall.c`
: This is a basic system call handler that implements the system calls mentioned above.

`lib/user/syscall.c`
: Provides library functions for user programs to invoke system calls from a C program. Each function uses inline assembly code to prepare the syscall arguments and invoke the system call. We do expect you to understand the calling conventions used for syscalls.

`lib/user/stdlib.S`
: Provides library functions for the C standard library linked into Pintos user applications, namely implementing `malloc`, `calloc`, `realloc`, and `free`.

`lib/syscall-nr.h`
: This file defines the syscall numbers for each syscall.

`userprog/exception.c`
: Handle exceptions. You are expected to modify `page_fault` the Pintos page fault handler, to complete this assignment. We already made some modifications to it in order to validate arguments to system calls.

You should carefully read through the functions in `threads/vaddr.h` and `userprog/pagedir.h`, as many of them will be useful to you in this assignment.

1.2.2 Pages

Allocating Pages

Use the following functions in `threads/palloc.h` to allocate and deallocate pages in the Pintos kernel.

```
void* palloc_get_page(enum palloc_flags);
void* palloc_get_multiple(enum palloc_flags, size_t page_cnt);
void palloc_free_page(void* page);
void palloc_free_multiple(void* pages, size_t page_cnt);
```

The `palloc` functions use a bitmap to keep track of which pages are free and which pages are allocated. The `flags` argument is a bitmask of the following choices.

```
enum palloc_flags {
  PAL_ASSERT = 001, /* Panic on failure. */
```

```
  PAL_ZERO = 002,   /* Zero page contents. */
  PAL_USER = 004    /* User page. */
};
```

The set of pages is partitioned into two separate *pools*: a user pool and a kernel pool. Pages in the kernel pool are meant for use inside the kernel (e.g. the stack for kernel threads) and pages in the user pool are meant to be mapped into the virtual address spaces of user processes. The reason for this separation is to prevent failures in the kernel if user programs run out of memory. The `PAL_USER` flag tells the `palloc` function to allocate the requested pages from the user pool. Otherwise, it will allocate the requested pages from the kernel pool. **If you intend to map a page into the virtual address space of a process, you should allocate it from the user pool using `PAL_USER`.**

Mapping a Page into a Virtual Address Space

The `pagedir` member of `struct thread` is a pointer to the page table of the process. When switching to a process, Pintos uses the `pagedir_activate` function to start using that process' page table for address translation, by setting the page directory base register (`%cr3`). Note that the entirety of kernel memory is mapped into every process' virtual address space at addresses `PHYS_BASE` and above, so all process' page tables are interchangeable as long as you are accessing kernel memory. For this reason, we sometimes refer to addresses at `PHYS_BASE` and above as *kernel virtual addresses*.

The physical address corresponding to a kernel virtual address can be computed by subtracting `PHYS_BASE` from it. In principle this need not be true, but Pintos sets up its page tables such that it holds. The functions `vtop` and `ptov` (`#include "threads/vaddr.h"`) are helper functions that perform this conversion for you, but we do not expect you to have to use these functions in this assignment.

Given a page allocated in kernel virtual memory, one can map it into the virtual address space of a process by calling `pagedir_set_page`, which handles traversal of the two-level hierarchical page table and allocation of leaves as needed. Two arguments to `pagedir_set_page` are the virtual address in the user process at which the page should be mapped, and the kernel virtual address of the page to map. **Although the physical page table entry contains the physical page number, remember to pass in the kernel virtual address into `pagedir_set_page`.** Once you've mapped the page into the process' virtual address space, you don't have to worry about deallocating it; when the process exits, the `process_exit` function will call `pagedir_destroy`, which will call `palloc_free_page` on all pages mapped into the process' address space. If you would like the page to continue to be allocated even after the process dies, you should first remove it from the page table using `pagedir_clear_page`.

Note that pages that you allocate using `palloc_get_page` could have been previously allocated and then freed, and therefore could contain data from the previous time it was allocated. If it was mapped into a user process, it could contain data from the previous user program. To properly enforce protection, **you should initialize the contents of a page before mapping it into a user process. This is typically done by setting all bytes in the page to zero**, except in special situations where the page should contain specific data (e.g. loading new code into a process or bringing back a page from disk). Under no circumstances should memory used by the kernel, or by other processes, become visible to a process because a physical page frame was reused.

Summary and Example

In summary, here is how to map a fresh page into the virtual address space of a process.

1. Allocate the page from the user pool using `palloc_get_page` and passing the `PAL_USER` flag.
2. Zero out the page's contents, either by using `memset` or passing the `PAL_ZERO` flag when allocating the page (e.g. `palloc_get_page(PAL_ZERO | PAL_USER)`).
3. Use `pagedir_set_page` to map the page into the virtual addres space of a process.
4. The page will be deallocated when the process exits and `pagedir_destroy` is called. Alternatively, if you would like the page to be deallocated at some other time, you can remove it from the page table with `pagedir_clear_page` and then deallocate it later using `palloc_free_page`.

A simple example of this is in the `setup_stack` and `install_page` functions in `process.c`. **We strongly recommend that you review these functions and understand this simple example before attempting this homework.**

2 Tasks

2.1 Stack Growth

In Project User Programs, the stack was a single page at the top of the user virtual address space, and programs were limited to that much stack. Now, if the stack grows past its current size, allocate additional pages as necessary. Allocate additional pages only if they "appear" to be stack accesses. Devise a heuristic that attempts to distinguish stack accesses from other accesses.

You may assume that a correct Pintos user program will never attempt to write to its stack below the stack pointer. The stack pointer will point at or below any active variables on the stack. In real operating systems, the system may interrupt a process at any time to deliver a "signal," which pushes data on the stack (or in kernel mode, an interrupt).[1] So, you should check if the faulting address is above the user value of `%esp` before extending the stack. However, there is an important edge case you need to handle. For example, the x86 `push` instruction checks access permissions before it adjusts the stack pointer, so it may cause a page fault 4 bytes *below* the stack pointer.[3] Similarly, the `pusha` instruction pushes 32 bytes at once, so it may fault 32 bytes below the stack pointer. Your implementation of stack growth should handle these cases.

You will need to be able to obtain the current value of the user program's stack pointer. Within a system call or a page fault generated by a user program, you can retrieve it from the `esp` member of the `intr_frame` passed to the system call handler or page fault handler, respectively. The provided code validates invalid arguments to system calls lazily, by terminating the user process in the system call handler for page faults to user memory addresses that occur when handling a system call.

Note that the page fault resulting in stack growth may not necessarily be for the next consecutive page on the stack—the stack may grow by multiple pages at a time. This leads to multiple ways to implement stack growth. Here are two possibilities:

1. Only map the page containing the faulting address, so that the set of mapped pages for the stack is not necessarily contiguous in the virtual address space.
2. Map *all* pages from the current stack until the page containing the faulting address, so that the set of mapped pages is contiguous in virtual addresses.

The first option is simpler, **but beware to handle the edge case where the first access to a stack page may occur while handling a system call.** For an example of this, see the `pt-grow-stk-sc` test. The reason this is an edge case is that, if a page fault occurs while handling a system call, the `esp` member of the `intr_frame` passed to the page fault handler does *not* contain the stack pointer of the user program (think about why). The value of the stack pointer in the user program at the time of the page fault is in the `intr_frame` passed to the system call handler. If your design requires you to consider stack extension due to a page fault while handling a system call, you will need to arrange a way to obtain the user-mode value of `%esp` in the faulting process in the page fault handler so that you can decide if the stack should be grown as a result of the page fault.

If there is no free page in the user pool to use to extend the stack, you should terminate the process with exit code -1. **Except for this out-of-memory condition, do not implement an absolute limit on stack size.** Modern operating systems do implement other limits, however; see the appendix (Resource Limits) for more information.

The first stack page need not be allocated lazily. You can allocate it at load time, as is done by the code provided to you in `process.c`, with no need to wait for it to be faulted in. However, future stack pages should only be allocated on demand. **It is not acceptable to simply allocate a large stack up front when the user program is loaded.**

[1] This rule is common but not universal. One modern exception is the x86-64 System V ABI[2], which designates 128 bytes below the stack pointer as a "red zone" that may not be modified by signal or interrupt handlers.

[3] If `%esp` were to be decremented by 4 bytes in the page fault handler, then the `push` instruction would not be restartable in a straightforward fashion.

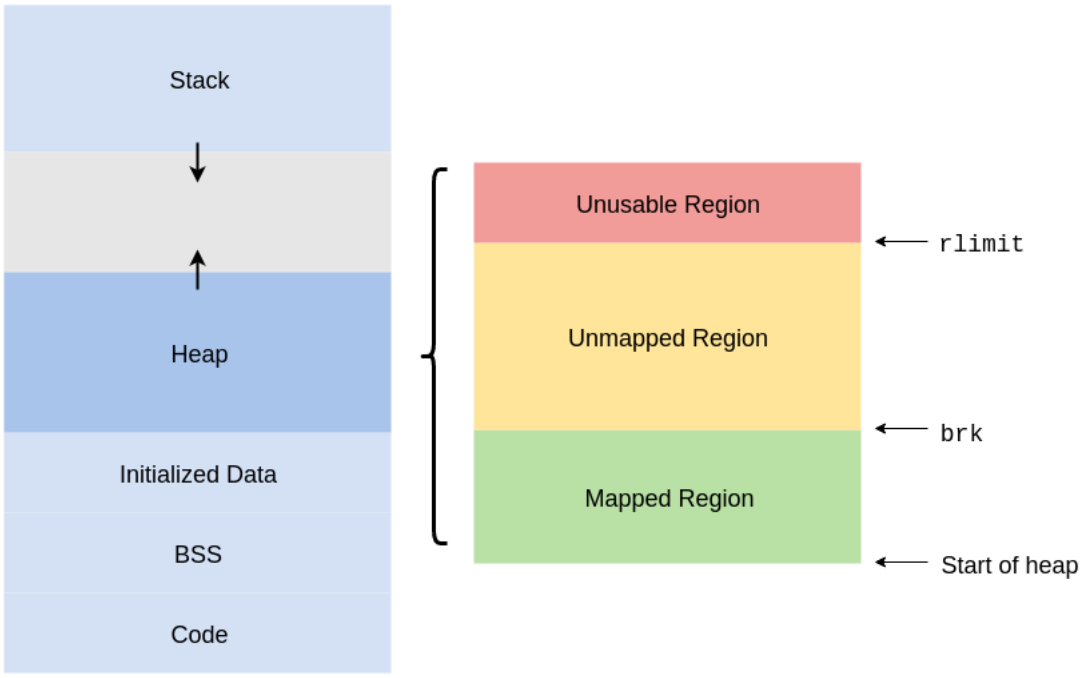
2.2 Dynamic Memory Allocation

For this portion of the homework, you'll need to extend Pintos with the `sbrk` system call, so that your dynamic memory allocator can request memory from the operating system.

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

2.2.1 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 the appendix (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.

2.2.2 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 the appendix for more information. See the appendix (Unmapped Region and No Man's Land) for more information.

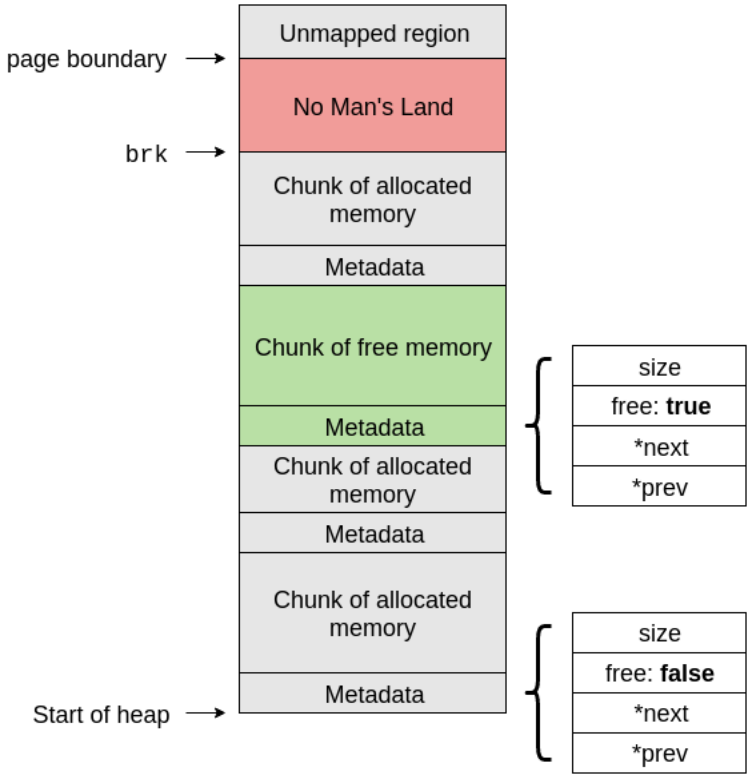
Allocating additional pages for a process' heap will fail if, for example, the user memory pool is exhausted and `palloc_get_page` fails. If `sbrk` fails, the net effect should be that `sbrk` returns `(void*) -1` and that the segment break and the process heap are unaffected. You might have to undo any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

A Unmapped Region and No Man's Land

We saw earlier that the break marks the end of the mapped virtual address space. By this assumption, accessing addresses above the break should trigger an error ("bus error" or "segmentation fault").

The virtual address space is mapped in quanta of pages (usually some multiple of 4096 bytes). When `sbrk` is called, the operating system will have to map more memory to the heap. To do that, it maps an entire page from physical memory to the mapped region of the heap. Now, it is possible that the break doesn't end up exactly on a page boundary. In this situation, what is the status of the memory between the break and the page boundary? It turns out that this memory is accessible, even though it is above the break and thus should be unmapped in theory. Bugs related to this issue are particularly insidious, because no error will occur if you read from or write to this "no man's land."



B Resource Limits

In Homework 0, you briefly explored the `getrlimit` syscall. In modern operating systems like Linux, processes have limits on their resource usage. For example, the maximum size of the stack and heap are governed by limits on the resources `RLIMIT_STACK` and `RLIMIT_DATA` (see `man 2 getrlimit`). Each of these resources has a *hard limit* and a *soft limit*. A process can raise its own soft limits; the soft limit exists to catch bugs (e.g. resource leaks) early by causing an error if a process uses more resources than expected. The hard limit can only be raised by the superuser (`root`), and exists to prevent resource abuse.

For this assignment, do not place an upper bound on the stack size or heap size. You do not need to implement resource limits.