Operating Systems Lab 4: Userland `exec()`

For the last lab of *Operating Systems*, you can choose between two assignments. Either, you can work on implementing background processes and signals in your existing shell, or you can work on implementing your own version of the `exec` system call you've become familiar with in the previous labs. Both options are similarly challenging, and can be tricky to get fully working - so make sure to start early and have plenty of time for debugging! Both assignments offer the opportunity to get bonus points. Note that switching between the assignments midway through is highly discouraged, in case you decide to switch after already making a submission to Themis, please send an email to the helpdesk, so we can make sure to grade the correct submission.

This document describes the `exec` assignment, for the last part of the shell assignment, refer to the other document on Brightspace.

Userland `exec()`

In this lab, you will be making your own version of the `exec()` system call. This version will live completely in userland, as opposed to the normal implementations that let the kernel handle the loading of the binary and memory management. For this lab, you will learn about how the kernel operates and deals with processes, and more specifically about memory management and security, ELF (Executable and Linkable Format) binaries and their format, the program stack, the ABI (Application Binary Interface), and a bit about linking.

A (small) word of warning beforehand: for an exercise like this, do not expect a simple Google search to turn up a fully-functional implementation - most of them are either very old and no longer work, could have never worked, or would only ever work in a very specific scenario. Copying such code will probably cost you more time in debugging than it will save you; instead, it is better to try and understand the assignment in detail, after which the code is actually quite straightforward. In addition, **it is important to read this entire document very carefully, and you should certainly not skip over the appendix!**

What the kernel does

As you might have already read in some online documentation, or in the man pages for `exec` (`man execve`) while you were working on the shell assignments, the `exec` system call will ask the kernel to start the execution of a given binary. The man pages contain an extensive description of all actions performed, however for the purposes of this assignment, it is only relevant to know that the `exec` system call asks the kernel to:

- Clear all memory allocated by the process calling `exec`,
- Load the given binary into memory,
- Set up a stack for the new binary, including values for `argc` and `argv`, and
- Start running the new binary.

What you will have to do

For this assignment, you will have to mimic the four actions typically done by the kernel, but now in userland, by writing the relevant C code. In the appendices of this document, you will find documentation on concepts such as the ELF file format, the ABI, and memory management. Not all information in the appendices will be directly necessary to write your code - most of it is there to give context to what you're working on, so that you gain a better view of the 'bigger picture'. In addition to this, *you will find a template on Brightspace that implements the tricky bits of this assignment.*

For now, let's get a bit more concrete. You should write a program `exec` (generated by a ≡ `Makefile`) that should accept the command to execute, including any parameters to that program, as arguments (so not on `stdin`!). Themis will therefore call your program as follows: `./exec /usr/bin/echo foo bar`. Your program should then execute the binary ≡ `echo` with arguments `foo bar`. All binaries will be provided as valid absolute or relative paths, as such they will be accepted as parameter to a `fopen()` call or similar functions and you do not have to search the `PATH`. Any contents provided on `stdin` should be untouched (i.e. you should not read from `stdin`), so that it can be consumed by the `exec`-ed program.

The following is a short outline of the code setup of your program. Any terms or concepts currently unfamiliar will be discussed in detail in the appendices to this document; on an initial reading this can be skipped.

1) Open the given binary, read the ELF headers and perform some sanity checks. For example, make sure the file is an ELF file and make sure that the values in the ELF header are usable for your application (for example, you cannot accept a binary compiled for ARM). You can use `fopen`, `fread`, `fseek`, and similar functions to work with the file.

2) Load all 'segments' of type `LOAD` into memory. This memory needs to be contiguous, in order to keep the relative pointers valid.

  a) For this, first determine the total span of all loadable segments (i.e. the start address of the first loadable segment until the end address of the last one) and allocate this amount of memory. Use `mmap()` for this.

  b) Fill the allocated memory with the binary contents properly, taking into account all offsets so that the binary 'structure' is preserved.

  c) Finally, protect the memory regions with the correct flags (as obtained from the program header) to enable execute- or write-access to the relevant regions.

3) `BONUS`: If the binary is dynamically linked, load the interpreter specified in the `INTERP` segment by repeating the steps above.

4) Construct a new, fresh stack for the new program with the exact structure as discussed in appendix C. An easy trick for saving some errorprone work is also discussed there.

5) Set the exit point (register `%rdx`), load the stack pointer, and jump to the entry point of the new binary (or the interpreter). Since this needs to be done in assembly, the implementation for this is already provided on Brightspace, but you will still need to call this code yourself.

During execution, your program should *print some tracing information.* When opening a binary, you should print `i Opening binary <name>`. When subsequently loading an interpreter, you should print `i Loading interpreter <name>`. The exit point function that is called after the program terminated should print `i Finishing up...` (three dots), and of course all these messages should be terminated by a newline.

As mentioned before, on Brightspace we will provide some template code. To be specific, we post a header file containing some helper macros for memory management and some code implementing the last step of the outline above (including printing the termination message). In addition, we post a header file with relevant `struct`s and constants for parsing ELF headers. Linux already ships one as well (`#include <elf.h>`), but our provided header is a bit more user-friendly with plenty of comments to assist you in understanding what's going on. Of course, you don't *have to* use the provided code and are free to build your own implementation entirely from scratch.

Lastly, as promised, this assignment has a bonus component: implementing support for loading interpreters and executing dynamically-linked binaries is optional and we will award **2 bonus points** for a fully-functional implementation. More details on the role of the interpreter can be found in appendix D.

Hints

As always, we will provide some hints for the implementation. These are quite technical and might only make sense to you once you actually need them.

- Most bugs in your implementation will manifest themselves as segmentation faults rather than clear error messages. So, whereas segmentation faults might be relatively rare in other assignments, you will probably experience many of them in this assignment. *This is expected.* Similarly, when Themis gives a segmentation fault, **this is most likely due to bugs in your code rather than Themis being wrong.**
- If you experience great difficulties and are not making progress, please try to run your code on a different platform. We tested our code on Ubuntu 20.04 and 22.04 (in a Docker container, and in WSL 1 and 2). Themis will run your code on Ubuntu 20.04.
- This assignment is specific to **x86-64** (so Intel x86 with the AMD 64-bit extensions)! Therefore, if you are working on any ARM-based platform (such as Apple M1), your code won't work natively. The easiest workaround is to run your code in an appropriate Docker container, for example by running `docker run -it --platform linux/amd64 ubuntu bash`.
- GDB and Valgrind will probably cause segmentation faults even when your code is working correctly. However, in our experience, when your code is working correctly the segmentation faults only occur *after* the `exec`-ed program *exits* - so if you get a segmentation fault at that point in your code while running GDB or Valgrind, it is probably safe to disregard.
- When you experience issues *before* the `exec`-ed program even starts, the best debugging tool is to (learn to) use GDB. Most editors (VS Code, CLion) have native integration with GDB for convenient debugging. To start debugging, it is advised to include `-g` in your ≡ `Makefile`, as this will generate 'debug symbols' which GDB uses to map assembly instructions to specific lines of code, so you can see what code is being run.

  However, a possible issue you might run into is that your program crashes right after the final jump to the entry point of the new binary. In that case, GDB does not know which binary you just loaded or where, and will not be able to display any information. You can resolve this by entering the command `add-symbol-file <filename> <address>` in the GDB debugging console, providing the filename of the binary you want to execute and the address at which you loaded the memory (which might be different every time depending on the return value of `mmap()`, so add a debugging print statement to recover that value).

  It will be useful to use the GDB console anyway, since most editors only allow line-by-line code advancements at best, while it can be very helpful to be able to advance by only a single assembly instruction every time (using `stepi`) - this way you can really check which jump or memory access causes an issue.
- You can use the `dumpstack` binaries we provide on Themis and Brightspace to check if your stack is set up correctly by comparing the results of running it directly or via your `exec`. Refer to the appendix for more information.
- All binaries we use for testing are relocatable code, either statically or dynamically linked. The testcase name on Themis indicated which is the case. The static binaries are compiled using `gcc -g -fpie -static-pie` and the dynamic ones using `gcc -g -fpie`. Using these commands you can compile your own binaries to test as well.

- In the ELF header, both dynamic and static binaries are marked with type `DYN`. This is due to the fact the code is relocatable. The historical reason of why this oddity came to be is widely documented online, in case you are wondering. So in the end, the only difference between dynamic and static binaries is the presence of the `INTERP` program header.
- Make sure to unbuffer your `stdout` to prevent issues on Themis. All test binaries will do this as well. So, just like in the shell assignment, call `setbuf(stdout, NULL)` at the top of your `main` function.
- When running dynamically-linked binaries, even properly functioning code won't print `i Finishing up...` at the end. This is because the interpreter that we load for running these binaries will override the exit handler.
- Be mindful of the fact that in C, pointer arithmetic is dependent on the pointer type. 'Incrementing' a pointer will add the size of the target type to the address: so `((uint32_t *) p)++` will add `4` to `p` (32 bits = 4 bytes), while `((uint64_t *) p)++` adds `8` to `p`. In general, a pointer of type `a` will move by `sizeof(a)` bytes per increment.
- When reading the ELF binary, you can just copy the bytes to a buffer and read it as an instance of `struct elf_header` directly - the data in the file already *is* an instance of this struct, so there is no need to do any manual manipulation. For example, you can use `struct foo *bar = malloc(sizeof *bar)` and then `fread(bar, sizeof *bar, 1, file)`.
- Please don't forget the option to go to the lab sessions and ask the TAs for help.

Good luck!

A Memory Management

It is important to know something about how Linux kernels typically do memory management before we can properly understand what is happening during an `exec` call. You might have seen aspects of this already in other courses or during the lectures, but here we will show an overview of the relevant concepts for this assignment. We will go a little bit in-depth as well, to explain how these concepts are typically implemented, beyond just telling you the bare minimum for finishing the assignment.

Virtual Memory

As introduced in the lectures, Linux uses virtual memory with paging. This means that every process gets its own (virtual) memory address space, completely empty[1] and isolated from any other process. This way, processes can be oblivious to the state of the rest of the system, and can let the kernel deal with managing the physical memory. For the purposes of this assignment, we are not bothered with the distinction between logical memory addresses (the ones inside the virtual memory space) and physical memory addresses. With the exception of this appendix section, when we refer to 'memory address' we mean a 'logical address', so an address valid inside the virtual memory space of the program.

For managing the virtual memory, Linux will use so-called (hierarchical) page tables. Page tables live in physical memory and describe the mapping between logical addresses and physical addresses. Each virtual memory space has an accompanying page table; one page table 'describes' the full 48 bits range of valid memory addresses[2] for a specific process.

Note that page tables are not a kernel feature, they are a platform feature: support for page tables is built-in on the x86 platform and each CPU will be able to natively use page tables for memory lookup (this is the job of the MMU). The kernel will set a CPU register indicating which page table to use, and then any subsequent memory access (read/write) will automatically be converted to the right physical memory address by the MMU. Similar techniques exist on other platforms, such as ARM and RISC-V, but the implementation details differ.

Memory Security

As mentioned before, the use of virtual memory on its own is already a security feature, as much as it is one for convenience. For further security purposes, the page table does not only store the mapping between logical and physical memory addresses, it will also store the permissions of each memory page: whether a given page has read-only, read-write, or read-execute access. In theory, it is possible to have RWX (so, read-write-execute) access, but typically the kernel instructs the CPU to disallow this combination.

This permission information is used by the CPU directly as well. When a program tries to access a specific (logical) memory address, not only will the MMU convert the address to a physical one, it will also check whether the access is intended for reading, writing, or execution, and will check whether the permissions allow for that. If this is not the case, the kernel is notified and this will then typically lead to the infamous segmentation fault.

[1] Empty except for the binary itself, and typically shared memory with the kernel.

[2] 32-bit processors unsurprisingly support 32-bit addressing, but current 64-bit processors 'only' support 48-bit addressing. The main reason for not covering the full 64 bits is space savings: the page table can simply be smaller and more efficient. Since 48-bits addressing for each virtual memory space already gives us 256TB of memory space *per process*, this will probably be plenty for the upcoming years. However, there is no inherent platform limitation that prevents expanding this range - there are already specifications extending to 57-bit addressing.

Memory allocation

Typically, you will use (variants of) `malloc()` in C to allocate some memory. These function calls are very convenient and performant, since they allow you to allocate chunks of any size and hide any memory management complexity. Meanwhile, behind the scenes in the kernel, various different memory allocation algorithms are used.

From a hardware perspective, the smallest chunk of memory you can ‘reserve’ is a single memory page: 4096 bytes. This corresponds to adding an entry to the page table. Of course, it would be crazy to allocate 4 kB for every string of a few characters you want to allocate: this is where more complex algorithms such as `kmalloc` or `vmalloc` come in.

These algorithms maintain a memory ‘pool’ in which the user can allocate smaller chunks of only a few bytes, limiting the overhead of memory allocations. The limitation of this approach is that we no longer have control over the permissions on these chunks of memory: the entire memory pool will always be read-write, and *that’s it*.

For the purposes of our `exec` implementation, however, we need to load a binary file that contains memory regions that are executable or read-only. Therefore, we need to allocate memory using `mmap`. This function will allocate memory directly in the page table for our process by reserving a number of pages, and we will then have full control over these. You can even request a specific (logical) memory address, but typically a call for ‘some memory region of size x’ will be sufficient.

Once we have allocated memory over which we have full control, we can set the proper permissions on this chunk of memory using `mprotect`. This function accepts a bitwise-or of flags for read, write, and execute. Note that since `mmap` and `mprotect` both interact with the page table directly, you can only allocate or protect multiples of the page size (4096 bytes) - you can’t allocate or protect ‘half a page’. And of course, if you allocate 10 pages, you are free to give different permissions to each of these pages separately.

B ELF File Format

Binaries in Linux are typically encoded using the ELF file format (Executable and Linkable Format) - `gcc` outputs such files. This format contains all the binary (assembly) code for a program to run, as well as instructions on how to run the code: where is the entry point of the code? What sections of memory should be loaded where? What sections of memory should be executable, and which should be writable? Are other files needed to run this binary? The answers to all these questions are contained in the ELF binary.

Magic bytes

To distinguish various file types, the kernel will look at the first few bytes of a file (so-called 'magic bytes'). For example, if these are `#!`, then the kernel knows this is a so-called 'shebang' and will know to start the binary mentioned as an interpreter for this file (as in `#!/bin/bash`). ELF files are identified with some other 'magic bytes', that you can find in the header file provided on Brightspace.

As an interesting aside: these file types and handlers are not hardcoded in the kernel and additional handlers can be registered dynamically[3] - even supporting file type detection based on filename extensions (like in Windows), instead of just 'magic bytes' at the start of a file. As a practical example, Microsoft added a special handler for Windows executable files ('WinPE binaries'), enabling users in WSL to 'start' `.exe` files that will then execute in Windows[4].

ELF Header

An ELF file typically contains four parts: the main header, the program headers, the data, and the section headers, in this order. The main header describes the properties of the file (library, executable, platform, bitness, endianness, etc.). The program and section headers both describe the binary contents from two different 'perspectives': the section view is primarily interesting for linkers, while the program view is relevant for execution - and therefore, for us. This is illustrated in figure 1.

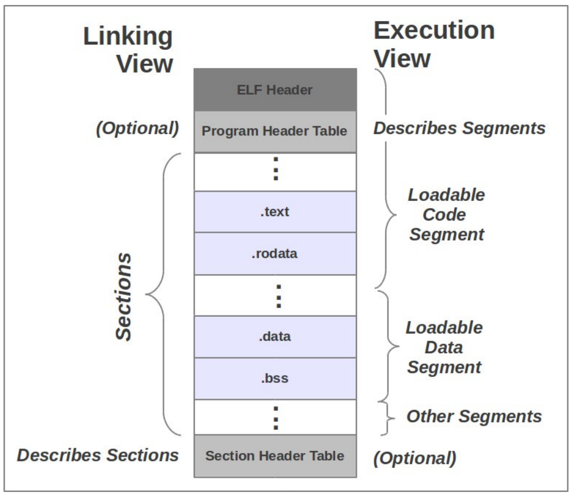

Figure 1: ELF binary structure, showing the two different 'perspectives' of the binary in terms of sections and segments.

Program Headers

The program headers each describe a 'segment', a part of the binary for which the header specifies what should be done with it (e.g. load or ignore), where it is located within the file, and where it should be located after loading to memory. Both the location in the file and the destination in memory are interpreted as pointers, and they are easy to confuse. Pointers like the first are what we call pointers in 'file space'. They are *relative to the start of the file.* So, this first pointer in the program header is a pointer in file space, it refers to where the contents of the current segment can be found *within the file.* Pointers like the second are in 'memory space'. They are pointers *relative to the start of our (virtual) memory space* - the 'typical' interpretation of a pointer as they just point to a location in memory.

[3]`https://docs.kernel.org/admin-guide/binfmt-misc.html`

[4]`https://learn.microsoft.com/nl-nl/archive/blogs/wsl/windows-and-ubuntu-interoperability#launching-win32-applications-from-within-wsl`

Typically, you will load the binary file into memory (e.g. using `fread`), after which you obtain a pointer to the beginning of the binary in memory. This allows you to translate a ‘file space’ pointer to a ‘memory space’ pointer: since file space pointers are relative to the start of the file, you can just add the memory address of the beginning of the file in memory to this ‘file space’ pointer, to obtain a ‘memory space’ pointer. This might be difficult to wrap your head around at first, so try to explicitly write out an example.

Program headers actually contain two fields of ‘where to put this segment’: a ‘physical address’ and a ‘virtual address’. Since we work in a virtual memory space (refer to the previous appendix), the physical address should be ignored and will typically be 0 anyway. The ‘virtual address’ is where the ELF binary expects us to load the segment into memory. Typically, these addresses are in the range of `0x40000` and above. It is important that these segments are loaded at the exact right location, since the actual assembly code inside expects itself to be there. For example, the ‘entry point’ defined in the main ELF header will point to a specific memory address that is only valid when you actually load the assembly code at the right addresses.

Relocatable code

The story from above is however a bit outdated. Nowadays, most binaries will contain only ‘relocatable’ code, which means that the code is not dependent on being loaded at specific memory addresses, so the binary can be loaded anywhere in memory. This is used for ASLR (Address Space Layout Randomization), a technique to limit the effectiveness of various memory attacks. Such a relocatable binary introduces yet *another* address space, the ‘binary space’. The binary contents will only contain relative pointers in this binary space, and similarly, the entry point in the ELF header is no longer an absolute memory address, but rather a (relative) pointer in binary space.

Let’s explain this using an example: say the binary you are trying to load is `0x200` bytes long (`0x` indicates a hexadecimal number). For relocatable code, the entry point might be something like `0x3F`. Then, if you load the binary into memory, you can do so at any memory location. We will use `mmap` to allocate some region of the right size, and we get back a pointer to the start of this region. For example, it could be `0x8F000`. Then, a pointer `0x16` in binary space will correspond to location `0x8F016` in memory. Similarly, the real entry point of our program will now be `0x8F03F`.

The surprising part might be how this will work in practice. Say you were to write some C code like `int *x = &foo;`, how will the compiler make this relocatable? It can’t say ‘ok well, variable `foo` will always be at location `0x5A`, so we will just store `0x5A` in `x`’, because the actual memory location is unknown to the compiler. It could do everything using relative pointers (‘`foo` will be `0x32` bytes before the current memory location’), but that is horribly complex. In practice, the kernel will pass the memory ‘offset’ (in our example, `0x8F000`) to the program as a parameter (refer to the ABI on how this is done), and then the C compiler will write assembly instructions that use this value to compute the memory location of `foo` at runtime.

For the scope of this lab, **we will only work with relocatable binaries**, so you do not have to worry about where your binary will be loaded - as long as the internal pointers are all consistent and the binary is loaded in one contiguous block. This will also prevent any conflicts where the new binary wants to load memory at an address already in use by your own `exec` binary. Whereas the kernel can just create a new, empty, virtual memory space, in userland you can’t do so, and you can’t (and shouldn’t) override your own memory either!

Example

To get some more insight in the ELF format, you can use the tool `readelf` that is provided on most Linux distributions by default. The flags `-h`, `-l`, and `-s` show the main header, program headers, and section headers, respectively. This tool is helpful to analyse existing binaries and their properties to determine how your code should behave.

Let's discuss a more concrete example: a simple hello world binary that is available on Themis. We run `readelf -l helloworld_static` and get output like the following:

```
Elf file type is DYN (Position-Independent Executable file)
Entry point 0x9f80
There are 12 program headers, starting at offset 64

Program Headers:
Type           Offset             VirtAddr           PhysAddr
FileSiz            MemSiz              Flags  Align
LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
0x0000000000008158 0x0000000000008158  R      0x1000
LOAD           0x0000000000009000 0x0000000000009000 0x0000000000009000
0x00000000000947f1 0x00000000000947f1  R E    0x1000
LOAD           0x000000000009e000 0x000000000009e000 0x000000000009e000
0x00000000000284c0 0x00000000000284c0  R      0x1000
LOAD           0x00000000000c6de0 0x00000000000c7de0 0x00000000000c7de0
0x0000000000005350 0x0000000000006a80  RW     0x1000

<omitted>
```

We can see the first few program headers with type `LOAD`, so we should load them into memory. The first program header says that the section of the *ELF file* from address `Offset` (`0x0`) of length `FileSiz` (`0x8158`) should be loaded to memory at some location `VirtAddr` (`0x0`), also with size `MemSiz` (`0x8158`). In our C code, we will be able to read a specific section of our file (using standard file operation functions), and we will have previously obtained (through `mmap`) a pointer to the start of the 'destination' memory region where we want to load our code (say, `0x400000`). We will then copy the contents from location `0x0` in the file to address `0x400000`. Similarly, for the second program header, we copy the contents from location `0x9000` in the file to address `0x409000`.

Note the gap in memory to load between the first two program headers[5]; the memory ranges are not guaranteed to be contiguous. Also be aware that the `VirtAddr` and `PhysAddr` values are not guaranteed to be aligned on a page boundary (i.e. a multiple of the page size). For example, the last program header shown above has a `VirtAddr` of `0xC7DE0`. However, you can only allocate and protect entire pages at a time - so make sure to handle this properly.

The `Align` parameter contains information on the expected alignment of the memory range. For our purposes, this field can be ignored. The `Flags` field is important, as it describes which memory page permissions should be set in the processor (see the previous appendix). Note that these permissions flags have different values than the ones `mprotect` expects - so make sure to manually convert between the values!

[5]The technical reason for this gap is that the two pages have different permissions: the first is read-only and the second should be read-execute. In the previous appendix we saw that these permissions can only be set on entire pages, so that's why the memory has a 'gap': we can only continue at the next page boundary (multiple of `0x1000`) when changing permissions.

C System V Application Binary Interface

In C, your program typically starts in the `int main(int argc, char *argv[]);` function. However, in other languages, there might not even be a main function, or it will have a completely different signature! This indicates that the `main()` function is in fact *not* the first thing that is being executed in a binary - it is not the 'entry point' as indicated in the ELF headers. If it would be, then the kernel would need to use different calling conventions for every language, which is not something we would want to do.

In practice, for a C program, the real entry point is a piece of assembly code (in C, labelled `_start`) that reads the 'kernel calling convention' and sets up the parameters for the `main()` function call properly. This kernel calling convention specifies how the kernel passes the program arguments and a bunch of other parameters to a user process, and it is part of the so-called 'System V Application Binary Interface'.

The System V ABI is a set of specifications governing interoperability between processes and between the kernel and processes in terms of function calls, system calls, file formats, and a range of other topics. It is adopted by Linux as default specification for how the kernel and processes should operate. For example: the ELF file format is part of the ABI. Relevant for our case is the specification of process initialization in section 3.4 of the x86-64 ABI (link below).

The process initialization specification mainly dictates the values of a few select registers and the layout of the stack of a fresh process. It specifies the register `%rsp` to point to the top of the fresh stack (so, the lowest address) which should be aligned on a 16-byte boundary. In addition, register `%rdx` should point to a `void` function that is called upon the exit of the binary (and it can be `NULL`).

Stack

As you might remember from previous courses, the stack typically contains the values of function-local variables, as well as the function call stack (this is where the terms 'stack overflow' and 'stacktrace' come from). On x86-64, the stack grows downwards, so the bottom of the stack is the largest address and is typically fixed, while the top of the stack is the lowest address and this value will change during execution.

The System V ABI dictates the following stack layout for new processes:

```
The memory addresses GO UP while we go down in this list - so we start with the lowest memory address
↪  and thus the top of the stack.
<name>            <size>   <description>
<< stack pointer points here, aligned on a 16-byte boundary >>

argc                 8     Number of arguments
argv[0]              8     Pointer to first argument
argv[..]            8*n    Subsequent argument pointers
argv[n]              8     NULL, marks end of arguments

envp[0]              8     Pointer to first environment variable
envp[..]            8*n    Subsequent env var pointers
envp[n]              8     NULL, marks end of env vars

auxv[0]              16    First auxiliary variable
auxv[..]             16    Subsequent auxvs
auxv[n]              8     NULL, marks end of auxvs

[ padding ]          >0    Fills the gap between the top and bottom of the stack;
                           everything before this is "top-aligned", and everything
                           after this is "bottom-aligned"
```

```
[ argv[i] strings ]  >=0    Where the argv[i] points to
[ envp[i] strings ]  >=0    Where the envp[i] points to

[ end ]               8     End of the stack, marked with NULL
<< this is the bottom of the stack, aligned on a 16-byte boundary >>
```

As you can see, the initial stack state contains more than just the `argc` and `argv` we might expect: it also contains the environment variables and the so-called 'auxiliary variables'. The environment variables can be obtained either through `getenv()`, or by adding `char *envp[]` as a third parameter to your `main()` function. The values of `argv` and `envp` are arrays of pointers that can in principle point anywhere in memory, but typically their values are stored at the very bottom of the stack in an 'information block'. These arrays do not have a fixed length and are `NULL`-terminated.

Note that the system expects your stack to be 'aligned on a 16-byte boundary'. This means that the highest address in your stack (so the bottom of the stack) should end with `0x...FFFF`, and the lowest address in your stack (top of the stack) should end with `0x...0000`. However, in practice this alignment on the top of the stack seems to not be very important.

On Themis, you can find a useful tool called `dumpstack`, which will display the entire stack. This is useful for examining the stack the kernel has set up for you, but also for examining your own code and checking whether you did the stack set-up correctly.

Auxiliary Variables

You are probably familiar with `argv` and environment variables, but you might not have heard of 'auxiliary variables' before. These are additional 'parameters' passed to your program with information about the platform, the process, and the ELF binary. These are sometimes also called 'ELF auxiliary variables', but they are not necessarily related to ELF.

Each of these auxiliary variables is a 16-byte value, with the first 8 bytes (64 bits unsigned integer) describing the type of the vector, and the second 8 bytes being the value. The possible types are documented in the ABI in section 3.4.3, and they include `AT_ENTRY` describing the process entry point, and crucially `AT_BASE` containing the base address at which the relocatable binary was loaded into memory. It is this value that allows relocatable code (see previous appendix) to work. The header files available on Themis include a struct definition for these auxiliary variables and an enum definition for the various types, to make it easier to deal with them.

Your Implementation

It can be very tricky to set up the stack for the newly spawned process correctly. Since the kernel has already set up a very nice stack for you, it is useful to re-use this stack for the `exec`-ed binary. You can allocate extra space at the top of the stack using `alloca()`, and write a new section of arguments, environment pointers, and auxiliary variables. Then you can consider the 'old' stack simply as an extra long 'padding' block, and re-use the information block (and thus the pointers pointing into this block).

Since your `exec` binary will be called with the new binary and its arguments as arguments for your code, the entire `argv` vector is already present in the existing stack - just with one additional element (`exec`) at the start. Is is therefore easy to reuse the string values for these arguments from the information block by simply copying the relevant pointers. Similarly, you don't want to change the environment variables, so you can again just copy all the pointers. For the auxiliary variables, you can copy most of them as well, but make sure to change the values of the ones that actually do need to change.

Try to think about this approach a bit, but be aware that `alloca()` allocates memory *on stack*. This means that if you do this inside a function, and then return from that function, all the memory is lost and your stack is reset to how it was before. Note as well that you should not overwrite any values of the existing stack - that might cause hard-to-debug issues.

D Linking

Programs rarely live on their own: they need to communicate with the kernel, they need to make use of the tools provided by the C standard library, or they need to interface with other programs. This is what linking is for: linking libraries and executables together. A given executable (or library) can be linked either statically or dynamically (or a mix of both).

A statically-linked executable is fully self-contained: it contains the code of all libraries it uses. This makes such an executable very portable and system-independent, however it is inefficient in terms of storage use and it might be difficult to patch security vulnerabilities in the standard library if then all dependent apps need to update as well.

A dynamically-linked executable, on the other hand, is linked at runtime. The binary only contains the main program code, together with information about which libraries it depends on. During runtime, the system will load the necessary libraries so that the program can run. Basically all common Linux tools use dynamic linking.

How does this work in practice? A static executable is easy: after loading it into memory, and setting up the stack as explained before, the code can just run immediately. However, a dynamic executable takes more work, as all the dependent libraries need to be loaded as well.

In practice, a so-called 'interpreter' is used: the kernel will start the interpreter, which will analyse the ELF binary and load the necessary libraries before starting the program itself. The ELF binary indicates which interpreter to use, through a special program header of type `INTERP` that points to a string with the library filename. This is also reflected in the output of `readelf -l` for a dynamic executable, in this case for ≡ `helloworld_dyn` that is also available on Themis:

```
Elf file type is DYN (Shared object file)
Entry point 0x31f0
There are 13 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000002d8 0x00000000000002d8  R      0x8
  INTERP         0x0000000000000318 0x0000000000000318 0x0000000000000318
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  <remaining headers omitted>
```

The kernel then not only loads the requested binary in memory, but also the interpreter. Instead of starting the requested binary at its entry point, the kernel will start the interpreter at *its* entry point, and inform the interpreter of the properties of the dynamic binary to be executed. This is done through the 'auxiliary variables' (see the previous appendix). For example, `AT_PHDR` contains a pointer to where the program headers are loaded in memory, `AT_ENTRY` contains a pointer to the entry point of the executable to load, while `AT_BASE` contains a pointer to the start of the memory section where the interpreter is loaded. A more detailed explanation of the variables can be found in section 3.4.3 of the x86-64 ABI, and in some of the other references below.

Implementing the userland `exec` for dynamically-linked executables will be a bonus for this assignment; you are only required to implement the static case as it is the easiest. However, once you have the static case up and running, supporting dynamic binaries is not too much work so we definitely encourage it!

E Useful sources

- ELF format specification
- Auxiliary vector information
- Thorough ELF introduction
- System V ABI for x86-64 (AMD64)
- Linux stack analysis[6]

[6]The code for the provided `dumpstack` binary is based on this, but adapted for 64-bit operation. However, this document is still very useful in understanding and interpreting the output of the stack dump.