CS350: Assignment 1 – Loading Programs

Emil Tsalapatis

For this course we will be writing code for a small but complete OS named CastorOS to implement real-world OS mechanisms. CastorOS is an educational OS that runs on real hardware or as a VM using virtualization [12]. In the course of the assignments we will write code to boot and run CastorOS, and then add critical system calls to it to implement important OS functionality like threading, scheduling, and file systems.

We will be running CastorOS using the QEMU emulator [1] that runs the entire OS inside a userspace process. QEMU emulates [4] real hardware using code, so the OS runs as if on bare metal. QEMU is available on Linux and MacOS, and on Windows through WSL [11].

In this assigment we set up CastorOS and fill in the code enough to make it boot into userspace. The OS is missing parts of the `Spawn` call that is needed to run applications and pass arguments to them. Booting up the system thus leads to a crash when the kernel tries to set up the `init` process. CastorOS's loader is incomplete and cannot load the program into the process. The loader also does not have code for copying the process' command line arguments into its memory. We will implement argument passing so that we can run userspace processes like shells.

1 Setting up the CS350 Client

We will be using the `client.py` command line tool to interact with the CS350 submission server. The tool is a script that we use to retrieve the source, submit our work, and check the status of our submission.

1.1 Setting Up the Submission Client

We first add our credentials to the `client.py` tool. To access the server we need to add our username and hex magic string to allow the script to authenticate with the server. The username is our 8-character UW ID. The magic hex string is a hex string that we receive via email from `cs350-noreply@uwaterloo.ca`. We directly modify in a text editor the `client.py` Python script provided in the course website. We change the values of the script's `STUDENT` and `MAGIC` variables from `None` to our username and magic string respectively. To test whether we have entered our credentials correctly we run:

```
$ python client.py ping
```

```
CS350 Server is active
$ python client.py status
Last Active Submission: None
Grades: None
```

We first use the `client.py ping` command to confirm that the submission server is accessible from our machine. After we ensure we can ping the server we run `client.py status` to ensure we have the proper credentials. If we have properly added our credentials to the client the server will respond with our last active submission and our active grade. In this case we have not submitted anything yet, so there are no active submissions or grades.

1.2 Downloading Castor OS

Next we download the newest version of the CastorOS source:

```
$ python client.py download
```

The command requests CastorOS's source from the submission server and places it in the current directory. We use `tar` to unpack the source into a CastorOS directory that we will be developing our solutions in.

```
$ tar zxvf castoros-latest.tar.gz
```

The starter code will be in the `castoros` directory.

2 Getting Started

We first install the SCons [8] build tool that compiles the OS and creates the runnable image. If you are on student.cs.uwaterloo.ca this is already done for you. The entire compilation process happens by running the `scons` command on the base OS source directory.

We must set SCons to use the LLVM [10] 15 toolchain to compile and link the image. OS images must conform to a very precise layout specification. Different compilers or even versions of the same compiler however generate different layouts for the same code and build arguments. LLVM 15 has been confirmed to produce a correct image so we will be using it for all assignments. We ensure that SCons uses LLVM 15 by creating a file called `Local.sc` in the CastorOS directory and directly defining the compiler version. We will be using the Clang compiler both for compilation and linking, so we add the lines `CC=clang-15` and `LD=clang-15` to `Local.sc`. Using the `cat` utility to inspect the file's contents should produce the following:

```
$ cat Local.sc
CC="clang-15"
LD="clang-15"
```

The Clang command may have a different name under different OSes. Please consult your OS's documentation for more details if running `clang-15` fails.

We start up CastorOS by passing the outputs of the compilation process to QEMU. The build process generates from the source tree a disk image in `build/bootdisk.img` and a kernel image in `build/sys/castor`. QEMU uses the disk image to emulate a disk device that behaves like a real disk. The disk image holds a file system with the userspace utilities, configuration files and user data. The kernel image only includes the kernel and is separate because it must be passed directly to QEMU for the machine to boot.

The kernel image conforms to the multiboot format [7] used by QEMU to set up the kernel for execution at boot time. This format is widely used by bootloaders [2]. CastorOS can thus be installed on real hardware using a popular multiboot compatible loader. We do not need to run CastorOS on bare metal for the base assignments.

We create the emulated machine where we will be running CastorOS using the following command from a terminal:

```
$ cd <path-to-basedir>
$ qemu-system-x86_64 \
-smp cpus=1 \
-kernel build/sys/castor \
-hda build/bootdisk.img \
-nic none \
-nographic
```

The above command creates the emulated machine and boots CastorOS. We use a single CPU for now (`-smp`) and pass the kernel image using its path in our local file system (`-kernel`). We use as a disk the image generated by the build process (`-hda`). The machine needs no network interface card (`-nic`). We use a serial console to communicate with the machine so we do not need a framebuffer [5] (`-nographic`). We either run the above QEMU command in the base source directory or adjust the arguments to the `-kernel` and `-hda` parameters accordingly.

We must ensure that the machine is booting properly before proceeding with the assignment. Entering `Ctrl+A X` kills the machine and frees up the terminal.

The output should look something like this:

```
SeaBIOS (version rel-1.16.2-0-gea1b7a073390-prebuilt.qemu.org)
Booting from ROM..Castor Operating System
Invalid magic number: 0x0
flags = 0x24f
mem_lower = 639KB, mem_upper = 129920KB
boot_device = 0x8000ffff
cmdline = /root/castoros/build/sys/castor mods_count = 0, mods_addr = 0x43b000
mmap_addr = 0x9000, mmap_length = 0xa8
 size = 0x14, base_addr = 0x0, length = 0x9fc00, type = 0x1
 <... omitted...>
Initializing GDT... Done!
Initializing TSS... Done!
Initializing IDT... Done!
Initializing Syscall... Done!
Initializing PMAP ... Done!
Initializing XMEM ... Done!
 <... omitted ...>
loader:   Offset            VAddr FileSize  MemSize
loader: 00000000 0000000000200000 00001744 00001744
loader: AllocMap 0000000000200000 00001744
loader: 00001750 0000000000202750 00004049 00004049
loader: AllocMap 0000000000202000 00004799
loader: 000057a0 00000000002077a0 00000200 000007a8
loader: AllocMap 0000000000207000 00000f48
loader: Jumping to userspace
CPU 0
Interrupt 14 Error Code: 0000000000000004
 <Debug Information>
Entered Debugger!
kdbg>
```

The goal of this assignment is to fix the OS so that it can finish booting. The diagnostics tell us that the OS properly initializes the hardware but crashes when it attempts to create the first userspace process called `init`. This is expected because the loader code is incomplete. In the next section we will go through the missing pieces we must write we to complete it.

The source tree includes the userspace, kernel, and tools for the OS. The tree looks as follows:

```
<path-to-basedir>
|--AUTHORS
|--bin
|--build
|--include
|--lib
|--LICENSE
|--pxelinux
|--release
|--sbin
|--SConstruct
|--sys
|--tests
```

The source tree holds the different userspace components and the kernel in separate directories. The kernel is entirely within the `sys/` directory and includes device drivers, system calls and core subsystems. Userspace programs are entirely within the `bin/` and `sbin/` directories. These programs use the system API defined within the headers in `include`. The `lib` directory holds userspace shared libraries like `libc` that implement the userspace part of this API and often interface with the kernel. The `build` directory is initially empty and serves as a destination for the compiled kernel binary and disk image.

The `Spawn` Call For this assignment we will be implementing the `Spawn` system call. `Spawn` creates a new process in the system that runs the application binary specified in the arguments. The call is a combination of the `fork` and `exec` system calls in UNIX-like systems. Windows has a system call with the same semantics called `CreateProcess` and modern UNIX-like systems support `posix_spawn`.

The complete signature of the system call is:

```
int OSSpawn(char *path, char *argv[]);
```

Spawn creates a new process and loads the binary specified by `path`, with an array of command line arguments specified by `*argv[]`.

The pseudocode for the `Spawn` call is the following:

Algorithm 2.1: Pseudocode for `Spawn`.

```
Input: Userspace address of program path string, userspace address of
       argument array
Output: Process ID of the new process
1  $path \leftarrow$ Copy_StrIn(userPath)
2  $args \leftarrow$ PAlloc_AllocPage()
3  copy in $userArgs$ from userspace into $args$
4  $elfhdr \leftarrow$ PAlloc_AllocPage()
5  get a file handle $fp$ from $path$
6  read in the first 1 KiB of $fp$ into $elfhdr$
7  $proc \leftarrow$ Process_Create()
8  $thr \leftarrow$ Thread_Create()
9  open the console and attach it to $proc$
10 $outbuf \leftarrow$ PmapTranslate()
11 Loader_Load($fp$, $elfhdr$)
12 copy out $args$ into $outbuf$
13 close the $fp$ handle
14 mark $proc$ as runnable
15 return $proc->pid$
```

We must write the code for the steps marked in red, while steps in black are already implemented. The call first copies in its arguments from userspace memory to kernel memory for later use. These are the path and an argument array that must be deep copied[3] into the kernel. `Spawn` allocates memory using `PMap_AllocPage` that for our purposes returns a kernel buffer. The system call then allocates a buffer for the ELF program header that we look into in the Section 3 and reads it from the program binary. `Spawn` creates metadata for the process and its thread and attaches the process to the console. Finally `Spawn` loads in the program itself into process memory, copies out to it the arguments passed by the caller, and marks it as runnable. Below we will go through the steps necessary to fill in the two gaps in the code.

3 Completing the Loader

Files to complete this section:

- `sys/kern/loader.c`
- `sys/kern/syscall.c` (you will complete this in the next section)

In this section we finish implementing the loader to allow the OS to load usespace processes. After completing this section the OS will load `/sbin/init` and then stop. In the next section, we will complete `spawn` to get the shell running.

Users run applications by invoking the `spawn` system call that loads a program binary and its arguments. For example, running `/bin/cat file1 file2` in our shell, calls the `/bin/cat` program with the arguments `[file1, file2]`.

The kernel first creates an empty process, with no data in its address space. It then reads in the program data from the binary and unpacks it into the address space. The kernel also initializes the CPU registers of the process thread so that it starts executing from `main` and copies the `argv` vector into the process. The resulting process is ready to run the program in userspace. The loader is the OS component that takes care of this address space and CPU initialization.

The loader should implement the address space and CPU initialization, but the version provided is missing the relevant code paths. The loader has two issues: It does not copy the program code into the process and does not set the initial instruction pointer. We will add the code for reading in the program binary, and set the initial instruction pointter by parsing the binary.

Reading in the program code is a simple `read` operation, but we must use the kernel virtual file system (VFS) API to implement it. The VFS API is defined in `sys/include/vfs.h` and describes the operations that can be done with file system files. We list the API below:

```c
int VFS_MountRoot(Disk *root);
VNode *VFS_Lookup(const char *path);
int VFS_Stat(const char *path, struct stat *sb);
int VFS_Open(VNode *fn);
int VFS_Close(VNode *fn);
int VFS_Read(VNode *fn, void *buf, uint64_t off, uint64_t len);
int VFS_ReadDir(VNode *fn, void *buf, uint64_t len, uint64_t *off);
```

The loader must parse the program binary and cannot just write it into the address as it is. CastorOS programs are in the ELF File Format [9] that is the standard binary format by modern UNIX-like systems. The ELF format describes programs as a set **sections** that comprise the program code and data. The ELF format also provides a set of **segments** through it's program headers that describe how to load the program into memory.

Each of these segments is mapped independently of the others and represents code, data, or zeroed out space (BSS). The loader's job is to iterate over the program headers in the ELF binary and read them into the new process. You can inspect the headers using the `readelf` utility in our local machine to print them out in human-readable form.

```
% readelf -h -l build/sbin/init/init
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 09 00 00 00 00 00 00 00 00
  Class:                             ELF64
...
  Entry point address:               0x202b00
...

Elf file type is EXEC (Executable file)
Entry point 0x202b00
There are 7 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flg    Align
...
  LOAD           0x0000000000000000 0x0000000000200000 0x0000000000200000
                 0x0000000000001af4 0x0000000000001af4  R      0x1000
  LOAD           0x0000000000001b00 0x0000000000202b00 0x0000000000202b00
                 0x0000000000004049 0x0000000000004049  R E    0x1000
  LOAD           0x0000000000005b50 0x0000000000207b50 0x0000000000207b50
                 0x0000000000000200 0x00000000000007a8  RW     0x1000
...
```

The listing above shows a truncated version of `init`'s ELF headers. The header includes the entry point address that points to the first instruction to execute. The headers below describe the sections of the address space that the loader will create and load the binary into. Note that the second segment has read/execute permissions and an address of `0x202b00`, the same as the entry point address. The two addresses are identical because the segment holds the program code that the process starts executing from.

We will use the `Lookup`, `Open`, `Read`, and `Close` calls to read in the segments. First we use `Lookup` to get a reference to the program file's handle using the file path passed by the user. We open the file for reading using `Open`, then `Read` the first 1 KiB of data. We already provide the buffer to be passed to `Read` as a variable called `pg`. CastorOS already has code create this buffer and attach it to the address space, so we only need to fill it with the ELF segment data.

The next step is to fix the `Loader_Load` call to traverse the list of segments and load the segments into the address space. The function already iterates over the program headers once to create the memory regions for each segment in the program's address space. Traverse all the program headers the same way that the existing loop does, loading the program code into the already created mapping. Only segments marked `PT_LOAD` must be loaded at load time.

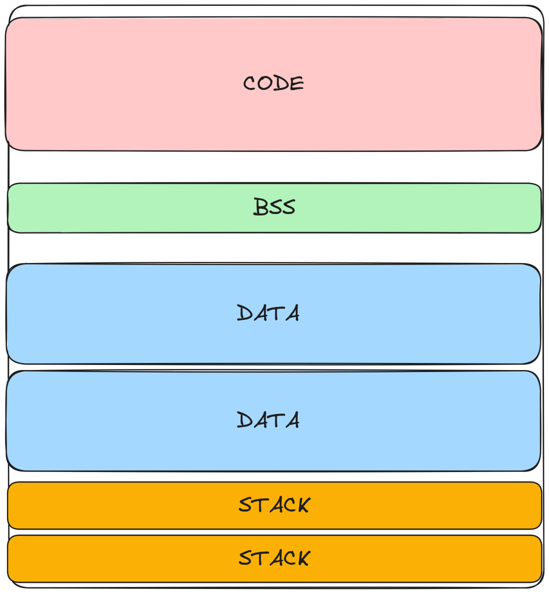

Figure 1: The memory layout of a userspace program. The code segment holds the executable program code. The BSS holds prezeroed variables, while the data segments are empty space created at runtime by the program itself. The kernel places the stack of each thread in the process in a dynamically created stack segment.

The ELF header has the following structure (code taken from Linux's `libc`):

```c
typedef struct {
        unsigned char e_ident[EI_NIDENT];   /* File identification. */
        Elf64_Half    e_type;        /* File type. */
        Elf64_Half    e_machine;     /* Machine architecture. */
        Elf64_Word    e_version;     /* ELF format version. */
        Elf64_Addr    e_entry;       /* Entry point. */
        Elf64_Off     e_phoff;       /* Program header file offset. */
        Elf64_Off     e_shoff;       /* Section header file offset. */
        Elf64_Word    e_flags;       /* Architecture-specific flags. */
        Elf64_Half    e_ehsize;      /* Size of ELF header in bytes. */
        Elf64_Half      e_phentsize;    /* Size of program header entry. */
    Elf64_Half      e_phnum;        /* # of program header entries. */
        Elf64_Half      e_shentsize;    /* Size of section header entry. */
    Elf64_Half      e_shnum;        /* # of section header entries. */
        Elf64_Half      e_shstrndx;     /* Section name strings section. */
} Elf64_Ehdr;
```

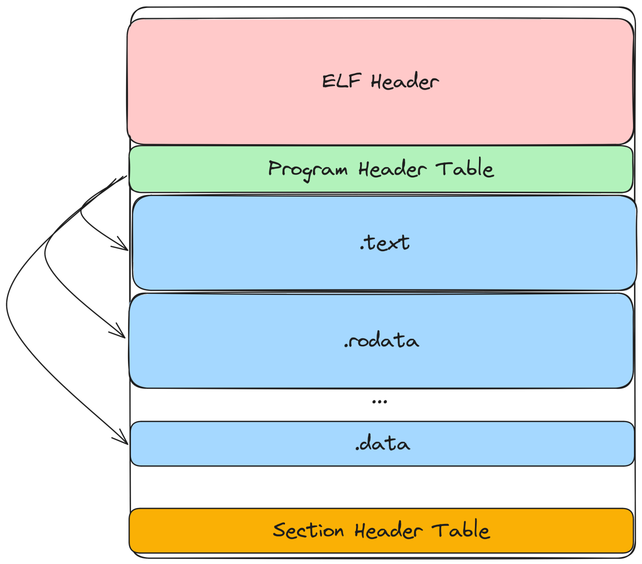

Figure 2: The ELF metadata of a program. In this assignment we only need to find the ELF header and program header table, then read in each segment.

CastorOS already has code for testing the integrity of the header, so we only need to parse the program headers in the file. First we access the `e_phoff` field in the ELF header that lists the offset in the file of the first program header. We also need `e_phnum`, the number of program headers in the ELF binary. Program headers have a fixed size and fits within the 1 KiB of the file we have read into the kernel, so we only need to typecast the data read from the file into an array of program headers. This conversion is already done in `Loader_Load` and provides us with the `phdr` pointer to the program header array.

To complete the function, we must iterate through the `phdr` array and load each segment into the address space. We will use the functions `LoaderLoadSegment` and `LoaderZeroSegment` from `sys/kern/loader.c` to read in and map segment into the process. Each header has the following structure:

```
typedef struct {
        Elf64_Word      p_type;         /* Entry type. */
        Elf64_Word      p_flags;        /* Access permission flags. */
        Elf64_Off       p_offset;       /* File offset of contents. */
        Elf64_Addr      p_vaddr;        /* Virtual address in memory image.
        */
    Elf64_Addr    p_paddr;      /* Physical address (not used). */
    Elf64_Xword   p_filesz;     /* Size of contents in file. */
    Elf64_Xword   p_memsz;      /* Size of contents in memory. */
    Elf64_Xword   p_align;      /* Alignment in memory and file. */
} Elf64_Phdr;
```

We use the helpers to load the segment data into the mapping, the zero out the rest. First we use `LoaderLoadSegment` to read the `p_filesz` of data from file offset `p_offset` into address `p_vaddr` of the address space. We then use `LoaderZeroSegment` to zero out the rest of the segment after calculating the location and size of the leftover region. We only load segments that of type `PT_LOAD` that means they are to be loaded at process creation time.

Implementing the loader should result in the system loading the init process. You will see an extra message in the console showing that init is attempting to load the shell that will then fail. Next we will complete `Syscall_Spawn` that will allow init to load the shell.

4 Passing arguments

Files to complete this section:

- `sys/kern/syscall.c`

The final step in the assignment is implementing argument passing. The current `spawn` call does not copy over any arguments to the `main` function of the process. Any shell command that uses arguments does not work properly. We will fill in the code in `Spawn` to remove this limitation.

`Spawn` is called from a userspace to create a new process. An example is a shell process that creates a new process for each command. When the user types a command in the command line, the shell responds by passing the arguments over to `Spawn`. The `Spawn` call creates a new process that runs then exits, after which the shell returns control to the user.

Let's take as an example an invocation of the `cat` shell command. The command opens all input files and prints their contents to the terminal. If file `a.txt` has contents `"Hello\n "` and file `b.txt` has contents `"World\n "`, then:

```
$ cat a.txt b.txt
Hello
World
$
```

The shell receives the arguments `[cat, a.txt, b.txt]`. The shell then resolves the full path of `cat` to `/bin/cat` and calls `spawn`, here called `OSSpawn`:

```
char *path = "/bin/cat";
char *args = { "cat", "a.txt", "b.txt" };
OSSpawn(path, args);
```

CastorOS already has code to read in the strings from user to kernel memory. The OS allocates a page into which it reads in all arguments as a sequence of strings. Each string is read using the `Copy_StrIn` functions that copies strings from userspace to the kernel. The code uses the page pushes each string into a buffer, so if we have arguments `"arg1\0", "arg2\0", "arg3\0"` in userspace the call produces a buffer with contents `"arg1\0arg2\0arg3\0"` where `\0` the null string termination character. The system call assumes there are up to 7 arguments and that each argument is at most 256 bytes long, so they fit in the 4 KiB buffer allocated in `Spawn`.

We must pass to the newly created process address space not just the arguments but also a way to find them. Part of the C runtime is `lib/libc/crt1.c` that parses the argument array given by the kernel and runs the program `main` function.

We map some process memory into the kernel to more conveniently copy over the arguments. By convention, the OS uses the addresses above the top of the stack as the location of the `argv` vector, so the userspace address of the buffer we will be storing the arguments in is `MEM_USERSPACE_STKTOP - PGSIZE`. The first 64 bytes of this buffer are used to store the `argv` vector itself, while the rest of the buffer holds the arguments.

We copy the data over using the buffer as a stack, pushing the string arguments one at a time. We first create an `uintptr_t *` array called `outarr` by typecasting the `argstart` pointer, then zeroing it out. We then treat the rest of the buffer as a `char *` buffer that we can copy the strings into. We also typecast the `arg` array that holds the copied over arguments and name it `inarr`. These typecasts make it easier for us to work with userspace addresses and help us avoid mistakes.

We then construct the `argv` vector in the user portion of memory and place its userspace address into the `argstart` variable. We go through the buffer that holds the concatenated arguments and use `strlen` to find the offset and length of each string in the buffer. Keep in mind that `strlen` does not include the null byte at the end of each string that must also be accounted for in the length. We then use `strcpy` to write the string into the output buffer. We modify the page in the kernel using direct assignment and `memcpy` because the code takes care of mapping the userspace page into the kernel. We treat the output buffer as a stack by keeping an offset in the buffer after which there is only free space.

Every time we add a new string we use write it at this offset of the buffer. We then adjust the offset by the length of the copied data, so that the next argument to be copied will be placed right after the data we just added. The entry at offset 0 of the argument vector holds the argument count `argc`, and does not hold a pointer to an argument.

Every time we copy over a string, we record the userspace address it will have at the right offset of the `argv` vector. We compute the userspace address by combining the offset of the string in the page with the base userspace address of the buffer. We then write the address into the `argstart` pointer array at same offset in the userspace array as the one it has in the in-kernel array. The `argstart` array is the `argv` array used by `main` in userspace, and is placed at

Algorithm 4.2: The argument copying algorithm.

```
Input: Userspace address of program path string, userspace address of
       argument array
Output: Process ID of the new process
1  $outarg \leftarrow$ (char *)argstart
2  $outarray \leftarrow$ (uintptr_t *)argstart
3  $inarray \leftarrow$ (uintptr_t *)arg
4  $offset \leftarrow$ the size of the 8-element uintptr_t array
5  Zero out $outarray$
6  for $i$ in [1..8] do
7      if $inarray[i]$ is NULL then
8          $outarray[0] \leftarrow i - 1$
9          break
10     $len \leftarrow$ the length of the new argument adjusted for \0
11     copy over the string to $outarg$ at the $offset$
12     $outarray[i] \leftarrow$ offset + the userspace address of $argstart$
13     $offset \leftarrow offset + len$
```

the beginning of the userspace page we store the arguments in.

If the loading and argument passing code is implemented correctly then the kernel properly boots into userspace and a shell prompt appears. We can enter commands like `ls` and `cd` to the prompt to explore the emulated machine's file system.

```
loader:   Offset            VAddr FileSize  MemSize
loader: 00000000 0000000000200000 00001744 00001744
loader: AllocMap 0000000000200000 00001744
loader: 00001750 0000000000202750 00004049 00004049
loader: AllocMap 0000000000202000 00004799
loader: 000057a0 00000000002077a0 00000200 000007a8
loader: AllocMap 0000000000207000 00000f48
loader: Jumping to userspace
Init spawning shell
syscall: Spawn(/bin/shell)
syscall: SPAWN ffff8100100098a0
loader:   Offset            VAddr FileSize  MemSize
loader: 00000000 0000000000200000 00001824 00001824
loader: AllocMap 0000000000200000 00001824
loader: 00001830 0000000000202830 000042a9 000042a9
loader: AllocMap 0000000000202000 00004ad9
loader: 00005ae0 0000000000207ae0 00000230 000007d8
loader: AllocMap 0000000000207000 000012b8
System Shell
```

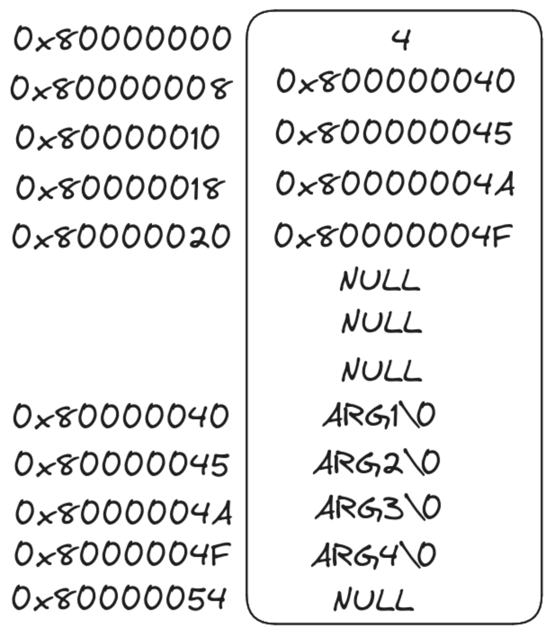

Figure 3: The layout of the arguments page in userspace. The first 8 words (64 bytes) are the argument array. The first entry holds the number of entries in the array, the rest hold either the userspace address of an argument or `NULL`. We place the argument strings themselves directly after the array. In this case our arguments are the four strings form `arg1` to `arg4` along with the string termination character `\0`. The strings are have size 5 and are stored consecutively in the page. The pointers in the array come in multiples of 5 for this reason.

5 Submitting Your Solutions

5.1 Submitting Results

After filling in the code we create a patch with our solutions. We first use `git commit` to create a single local commit with our additions to the codebase. **The commit should only include the sys/kern/syscall.c and sys/kern/loader.c. If the commit includes any other files the server will automatically reject the submission.** Next we use `client.py patch` to generate a patch out of our local source tree:

```
$ python client.py patch
```

We can provide the script with the location of the source tree using the `--srcroot` flag. If such an argument is not provided, the script defaults to searching for the `castoros` source tree in the current working directory.

```
$ python client.py patch --srcroot /path/to/castoros/dir
```

The last step is to submit the patch to the submission system:

```
$ python client.py submit
```

The patch should be called "castoros-patch" and be in the current working directory. The command sends the patch over to the submission server and queues it for evaluation. **The submission server takes about an hour to evaluate submissions. Submitting a new patch overwrites any active submissions without evaluating them.**

We can monitor the status of our submission using the `client.py status` command we saw earlier:

```
$ python client.py status
```

6 Debugging CastorOS

For debugging our solution we use three tools: `kprintf` debugging, the `kgdb` kernel debugger from inside the OS, and the `GDB` from outside the OS. We choose which tool to use depending on the nature of the bug we are tracking down.

The simplest technique at our disposal is `kprintf` debugging. We add `kprintf` statements in the kernel at various points in the code to print helpful diagnostics throughout execution. Using `kprintf` is quick and easy but requires changing the code every time we want to move or change a print message. Using `kprintf` is also not always possible, e.g., during early boot.

Make sure to remove all `kprintf` messages from your code before you submit. The grading scripts used to evaluate submissions read the serial console of the QEMU machine to check whether basic shell commands like `ls` and `cat` work correctly. Leftover `kprintf` messages may write to the console when these commands are being executed and cause the submission to fail the script's tests even if it is correct.

Another tool at our disposal is the standard GDB debugger. QEMU allows GDB to directly hook into the running OS and debug it as if it was a regular process. To use GDB with CastorOS we first add the `-s` and `-S` flags when invoking QEMU [6]. These flags will make QEMU listen for connections from a local GDB instance and also prevent it from running the OS immediately.

If you want debug symbols you will need to change the BUILDTYPE to DEBUG by editing your Local.sc and adding BUILDTYPE="DEBUG" to it. I would recommend removing this when you don't need it.

```
cqemu-system-x86_64 -s -S -nic none -m 64 -smp cpus=1 -nographic \
    -kernel build/sys/castor -hda build/bootdisk.img
```

We use GDB by running the following command from another terminal:

```
(gdb) target remote localhost:1234
```

This command will attach the GDB instance into CastorOS running inside QEMU. Running

```
(gdb) continue
```

will allow CastorOS to start booting. We then debug our OS as if it were a regular application. Please refer to one of the many GDB tutorials out there for details on how to use GDB for debugging.

The third tool is CastorOS's builtin `kgdb` kernel debugger. The main advantage of `kgdb` is its direct access to kernel state. CastorOS enters `kgdb` either if there is a kernel crash or if we run the `bkpt` command from the CastorOS terminal. If we enter `kgdb` using `bkpt` we resume execution with `continue`.

Running `help` in `kgdb` gives us a list of commands we can run. Each command gives us information about a certain part of the system, e.g. CPU state or running processes/threads. Using `kgdb` is ideal for bugs that do not crash the system immediately but eventually cause problems or crashes.

7 Text questions

- What is the difference between `fork`, `exec`, and `spawn`? Are all three necessary in a system?
- How does `fork` create a new copy of program memory in UNIX-like systems?
- In the ELF format `PT_DYNAMIC` is used to denote a dynamically linked segment. What does this mean? How is dynamic linking useful?

References

[1] , August 2023.

[2] Bootloader. https://en.wikipedia.org/wiki/Bootloader, August 2023.

[3] Deep Copy. https://developer.mozilla.com/en-US/docsc/Glossary/Deep_copy, August 2023.

[4] Emulator. https://en.wikipedia.org/wiki/Emulator, August 2023.

[5] Framebuffer. https://en.wikipedia.org/wiki/Framebuffer, August 2023.

[6] GDB usage. https://qemu-project.gitlab.io/qemu/system/gdb.html, August 2023.

[7] Multiboot Specification. https://www.gnu.org/software/grub/manual/multiboot/multiboot.html, August 2023.

[8] SCons: A software construction tool. http://scons.org, August 2023.

[9] The ELF File Format. https://wiki.osdev.org/ELF, August 2023.

[10] The LLVM Compiler Infrastructure Project. https://llvm.org, August 2023.

[11] What is the Windows Subsystem for Linux? https://learn.microsoft.com/en-us/windows/wsl/about, August 2023.

[12] What is Virtualization? https://aws.amazon.com/what-is/virtualization, August 2023.