CS350 Operating Systems Winter 2022

A3 - Assignment Specification

1 Introduction

The assignment is divided into two major components. The first requires implementing additional functionality to support argument passing to the kernels very first program that runs, i.e., to the program that is launched in response to the `"p''` (runprogram) kernel command and to implement a new system for OS/161 `execv`. The second requires a replacement of the virtual memory system. OS/161 has a very simple virtual memory system, called `dumbvm`. Assignment 3 is to replace dumbvm with a new virtual memory system that relaxes a few of dumbvm's limitations.

For this assignment, you are expected to do complete the following tasks:

1. Ensure that it is possible to pass arguments to the first process by modifying the kernel's `runprogram` command to support argument passing.
2. Implement the OS/161 `execv` system call, including argument passing.
3. Handle full TLBs
4. Change how address spaces are loaded, so that each application's `text` segment is read-only.
5. The dumbvm virtual memory system has two severe limitations in the way it manages physical memory.First, it assumes that each segment will be allocated contiguously in physical memory. Second, it never re-uses physical memory. That is, when a process exits, the physical memory that was used to hold that process's address space does not become available for use by other processes. As a result, the kernel will quickly run out of physical memory. Your objective in this assignment is to remove both of these limitations. In particular:
   - It should be possible for the pages in process' address spaces to be placed into any free frame of physical memory. That is, the kernel should no longer require that address space segments be stored contiguously in physical memory.
   - When a process terminates, the physical frames that were used to hold its pages should be freed, and should become available for use by other processes.

Your implementation should correctly and gracefully handle error conditions, and properly return the error codes as described on the man page. This is because application programs, including those used to test your kernel for this assignment, depend on the behaviour of the system calls as specified in the man pages. **Under no circumstances should your kernel to crash.**

All code changes for this assignment should be enclosed in `#if OPT_A3` statements. Don't forget to add `#include "opt-A3.h"` at the top of any file for which you make changes for this assignment.

In this assignment, you will implement a new system call, `execv`. The `execv` manual page specifies how the second parameter (the argument array) must be set up. Make sure you understand this before proceeding.

Argument passing means taking the arguments that are passed to `execv` and making them available to the new program that will start running in the process that does the `execv`. To do this, your kernel will need to retrieve these arguments from the address space of the original program (before destroying its address spaces) and then set up a properly structured argument array in the address space of the new program before it starts running. You will need to explore where in the new address space to place the arguments.

In addition to passing arguments to new programs through `execv`, your kernel is also expected to be able to pass arguments to the very first program that runs, i.e., to the program that is launched in response to the "p" (runprogram) kernel command. This is similar to passing arguments through `execv`, except for the fact that the arguments are coming directly from the kernel (which reads them from its command line) rather than from the program that is making the `execv` call.

Important: before you start working on this assignment, you should reconfigure and rebuild your OS/161 kernel.

In the `linux.student` environment, this is done by:

```
cd kern/conf
./config ASST3
cd ../compile/ASST3
bmake depend
bmake
bmake install
```

All OS/161 kernel builds for this assignment should occur in the `kern/compile/ASST3` directory in the `linux.student` environment.

In the `cs350-container`, you can configure and build the kernel by:

```
build_kernel ASST3 //configure and build
cd /os-compile/
sys161 kernel-ASST3 //run the kernel
```

Some general advice for this assignment:

- **Start early.** The instructions are detailed, but even debugging simple mistakes are time consuming. This holds doubly true if you are not especially familiar with `C`.
- **Compile often.** By checking whether the code compiles after every programming prompt, explicitly specified in the following sections, you will be able to pinpoint problems very quickly. The code **should** compile after every properly implemented programming prompt.

In this assignment, for the A3 - OS/161 kernel programming componenet, we will use two prompts.

Explore prompts: guide you towards a better understanding of the OS/161 kernel code.

Programming prompts: give you step by step implementation instructions for implementing synchronization primitives in OS/161 kernel. **Read these carefully!**

2 Overview of Requirements

This assignment is composed on the following tasks: In the first part you will implement argument passing for the `runprogram` function. This function implements process creation for commands passed to the kernel through the command line. The function currently does not accept arguments, so commands of the form `p <command>;q` can only be passed the name of the executable.

The second part includes implementing the `execv`. This system call is very similar in functionality to `runprogram`, but uses an existing process to execute the new program. This difference makes it necessary to clean up past process state and ensure we properly pass the arguments from the old to the new address space.

The third part of the assignment includes expanding TLB functionality. Right now each process uses each of the TLBs slots in succession and is unable to replace past entries with new ones.

Currently, the dumbvm system will panic and crash if the TLB fills up with valid entries. Change this so that the kernel will not panic in this situation. If kernel needs to add a new entry to the TLB and the TLB is full, the kernel should simply overwrite one of the existing entries with the new entry. A simple way to do this is to use the tlb random function to allow the hardware to choose a random entry to be overwritten. The `vm_fault` function is responsible for managing the TLB, so you should be able to implement this part of the assignment with some small changes to that function.

Furthmore, in the dumbvm system, all three address space segments (text, data, and stack) are both readable and writable by the application. For this assignment, you should change this so that each application's `text/` segment is read-only. Your kernel should set up TLB entries so that any attempt by an application to modify its text section will cause the MIPS MMU to generate a read-only memory exception Thus, we cannot ensure that the process does not accidentally overwrite `text` segments.

Finally, the assignment includes building a simple physical page allocator. Right now the kernel essentially uses a bump allocator to provide pages, without implementing any kind of freeing. The system thus runs out of memory very quickly even when lightly loaded. We will build a bitmap-based allocator that allows us to run indefinitely.

3 Code Review

This section gives a brief overview of some parts of the kernel that are relevant to the new Assignment 2b requirements.

3.1 `kern/syscall`

This directory contains the files that are responsible for loading and running user-level programs, as well as basic and stub implementations of a few system call handlers.

`loadelf.c`: This file contains the functions responsible for loading an ELF executable from the filesystem into an address space. (ELF is the name of the executable format produced by cs350-gcc.)

`proc_syscalls.c`: This file is intended to hold the handlers for process-related system calls, including the handler for `execv`, which you are implementing for this assignment.

`runprogram.c`: This file contains the implementation of the kernel's `runprogram` command, which can be invoked from the kernel menu. The `runprogram` command is used to launch the first process run by the kernel. Typically, this process will be the ancestor of all other processes in the system. Studying the `runprogram` function should give you some ideas on how to implement `execv`. Think about how `runprogram`'s task is similar to `execv`'s, and how it is different.

3.2 `kern/arch/mips/`

This directory contains machine-specific code for basic kernel functions, such as handling system calls, exceptions and interrupts, context switches, and virtual memory.

`syscall/syscall.c`: This file contains the system call dispatcher function, called `syscall()`. As was described in Assignment 2a, you will need to modify this function to invoke your handler for `execv()`.

`locore/trap.c`: In this file, in addition to the kernel exception handler, you will find the function `enter_new_process`, which should be useful for your implementation of `execv`.

`vm/dumbvm.c`: This file contains the machine-specific part of OS/161's very simple implementation of virtual address spaces.

`vm/ram.c`: This file includes functions that the kernel uses to manage physical memory (RAM) while the kernel is booting up, before the VM system has been initialized. Since your VM system will essentially be taking over management of physical memory, you need to understand how these functions work.

`include/tlb.h`: This file defines the prototypes for the kernel's functions for managing the TLB (such as tlb write), as well as definitions of the fields in each TLB entry.

`include/vm.h`: This file defines some macros and constants (such as the page size) related to address translation on the MIPS. Note that this `vm.h` is different from the `vm.h` in `kern/include`.

3.3 kern/vm

The `kern/vm` directory contains the machine-independent part of the kernel's virtual memory implementation.

`copyinout.c`: This file contains functions, such as `copyin` and `copyout` for moving data between kernel space and user space.

`kmalloc.c`: This file contains implementations of kmalloc and kfree, to support dynamic memory allocation for the kernel.

`uw-vmstats.c`: contains code for tracking statistics related to the virtual memory system. Not used for this assignment.

3.4 kern/include

`vfs.h`: This file describes the VFS interface, which the kernel uses to open and close files, e.g., program executable files. See the `runprogram` function for an example of how to use the VFS interface.

`vnode.h`: Opening a file using the VFS interface results in a `vnode` object representing the open file. The `vnode` object can be used to read data from and write data to the open file. `vnode.h` describes the `vnode` interface. See `loadelf.c` for an example of code that uses `vnode` operations to read data from a file.

`uio.h`: This file describes the `uio` interface. The kernel uses `uio` structures to describe a data transfer between a file and memory, between memory and a file, or between two locations in memory. `vnode` operations, such as `VOP_READ`, expect `uio` structures as parameters.

`addrspace.h`: Defines the addrspace interface. You may need to make changes here, at least to define an appropriate addrspace structure.

`vm.h`: Some VM-related definitions, including prototypes for some key functions, such as vm fault (the TLB miss handler) and alloc kpages (used, among other places, in kmalloc).

4 Adding argument passing to runprogram

The function `runprogram` in `kern/syscall/runprogram.c` is used to create new processes in the `OS 161` kernel. We create new processes from the menu by entering commands.

4.1 The boot process

Before we look at `runprogram`, let's see how we actually get to the shell prompt first. The kernel boot process in `OS 161` is very straightforward. Let us inspect it, by finding the `main` function:

```
root@os161:~/cs350-os161/os161-1.99# grep -r main \kern
kern/synchprobs/catmouse.c: * Values for these parameters are set by the main driver
kern/synchprobs/catmouse.c: * Once the main driver function (catmouse()) has created the cat and mouse
kern/synchprobs/catmouse.c:  /* create the semaphore that is used to make the main thread
kern/synchprobs/catmouse.c:  /* launch any remaining mice */
kern/synchprobs/traffic.c: * Values for these parameters are set by the main driver
kern/synchprobs/traffic.c: * Once the main driver function has created the
kern/include/vnode.h: *                      This maintains the open count so VOP_CLOSE can be
kern/include/uio.h:     size_t            uio_resid;    /* Remaining amt of data to xfer */
kern/include/test.h:/* The main function, called from start.S. */
kern/include/test.h:void kmain(char *bootstring);
```

Note, that there is no `main` function, but there is a `kmain`. Let's look for that:

```
root@os161:~/cs350-os161/os161-1.99# grep -r kmain \kern
kern/include/test.h:void kmain(char *bootstring);
kern/arch/sys161/startup/start.S:   jal kmain
kern/arch/sys161/startup/start.S:    * kmain shouldn't return. panic.
kern/arch/sys161/startup/start.S:   .asciz "kmain returned\n"
kern/startup/main.c:kmain(char *arguments)
```

So the function is defined in `kern/startup/main.c` and used in `kern/arch/sys161/startup/start.S`, which is an assembly file. This gives us strong hints that it holds the bootstrapping code. Inspecting the file shows we are correct - in it we define the `__start` symbol, used to identify the main function of the final binary. This will be set up by the system bootloader before this bootstrapping function is executed:

```
   .set noreorder

   .text
   .globl __start
   .type __start,@function
   .ent __start
__start:
```

The bootstrapping code copies over the bootstring with any argument for the kernel into the initial CPU's stack. It then copies the exception handling code to the physical addresses expected by the architecture. The code then initializes the TLB, sets up coprocessor 0 to specify that we are in kernel mode, and calls `kmain`. If everything goes well, we never return. Note that since the startup code is in assembly, we are making calls to C code using the `jal` instruction and adhering to the ABI.

The argument we pass it is the bootstring we specify in the command line. This is how commands are passed from the host's shell to the emulator's kernel. If we do not enter `q` in the initial command string, then the system directly reads our commands.

```
void
menu(char *args)
{
        char buf[64];

        menu_execute(args, 1);

        while (1) {
                kprintf("OS/161 kernel [? for menu]: ");
                kgets(buf, sizeof(buf));
                menu_execute(buf, 0);
        }
}
```

The `kmain` function calls `boot`, which sets up kernel subsystems, and `menu`. The menu function is just a loop that executes first the commands in the bootstring, then any commands we pass to its terminal afterwards. Since for testing we pass to the kernel commands in the format `p <program>; p <program>; ... ; q`, `menu_execute` runs first.

```
static
void
menu_execute(char *line, int isargs)
{
        char *command;
        char *context;
        int result;

        for (command = strtok_r(line, ";", &context);
             command != NULL;
             command = strtok_r(NULL, ";", &context)) {

                if (isargs) {
                        kprintf("OS/161 kernel: %s\n", command);
                }

                result = cmd_dispatch(command);
                if (result) {
                        kprintf("Menu command failed: %s\n", strerror(result));
                        if (isargs) {
                                panic("Failure processing kernel arguments\n");
                        }
                }
        }
}
```

The `menu_execute` function simply parses the bootstring into multiple command strings and calls `cmd_dispatch` for each of them. The function in turn tokenizes each command and checks a dictionary for the C function corresponding to the string. It passes all the arguments contained in the command string to the command. This is why expanding `runproram` is easy; its caller is already passing it user arguments, we just need to add code to handle them.

```
int
cmd_dispatch(char *cmd)
{
    ...
    for (word = strtok_r(cmd, " \t", &context);
            word != NULL;
            word = strtok_r(NULL, " \t", &context))
    ...
    for (i=0; cmdtable[i].name; i++) {
            if (*cmdtable[i].name && !strcmp(args[0], cmdtable[i].name)) {
                    KASSERT(cmdtable[i].func!=NULL);

                    gettime(&beforesecs, &beforensecs);

                    result = cmdtable[i].func(nargs, args);
                    ...
    }
    ...
```

The command table is simply an array of string,function pairs. The code defines an anonymous struct in C containing the pair, then an array of the struct's instances called `cmd_table`:

```
static struct {
        const char *name;
        int (*func)(int nargs, char **args);
} cmdtable[] = {
        /* menus */
        { "?",          cmd_mainmenu },
        ...
        /* operations */
        { "s",          cmd_shell },
        { "p",          cmd_prog },
        ...
        { "q",          cmd_quit },
        ...
```

Here, we can find which in-kernel `C` functions correspond to all the commands we've been using throughout the course. Since we are implementing argument passing, we need to look into `cmd_prog`, which passes all arguments to `common_prog`. With `common_prog` we are finally going from parsing to creating the new process:

```
static
int
common_prog(int nargs, char **args)
{
        ...

        /* Create a process for the new program to run in. */
        proc = proc_create_runprogram(args[0] /* name */);
        if (proc == NULL) {
                return ENOMEM;
        }

        result = thread_fork(args[0] /* thread name */,
                        proc /* new process */,
                        cmd_progthread /* thread function */,
                        args /* thread arg */, nargs /* thread arg */);
        ...
}
```

If you remember from Assignment 1, this is exactly the kind of work we do in `sys_fork`! The only difference is that instead of passing a trampoline function to `thread_fork`, we are passing `cmd_progthread`. That function is a wrapper for `runprogram` that copies over the function name into a local buffer then passes the function name to the process.

Let's now look at the function itself. We'll go through it piece by piece. First, the function opens the file of the executable passed to it. The goal is to dump the code into the address space of the new process so that userspace can run it.

```
int
runprogram(char *progname)
{
        ...
        /* Open the file. */
        result = vfs_open(progname, O_RDONLY, 0, &v);
        if (result) {
                return result;
        }
        ...
```

The next step is to actually create the address space. Remember that we are currently running as a kernel thread created from the kernel, and do not have an associated address space. When we `fork` the new kernel thread would run in the old process' address space before it created its own. Since here the thread was created directly by the kernel, there was no address space to inherit.

```
        /* We should be a new process. */
        KASSERT(curproc_getas() == NULL);

        /* Create a new address space. */
        as = as_create();
        if (as ==NULL) {
                vfs_close(v);
                return ENOMEM;
        }

        /* Switch to it and activate it. */
        curproc_setas(as);
        as_activate();
```

Now that we have an address space, we can dump the ELF executable into it. We close the file after we are done.

```
        /* Load the executable. */
        result = load_elf(v, &entrypoint);
        if (result) {
                /* p_addrspace will go away when curproc is destroyed */
                vfs_close(v);
                return result;
        }

        /* Done with the file now. */
        vfs_close(v);
```

Having defined the code and data segments for the new process, we need only define the stack pointer. After we do so we can jump to the new process, much like we do after fork. Instead of using `enter_forked process`, like with `fork`, we call `enter_new_process` and provide the address of `main`. The function will set up the program counter and store the userspace addresses of the command line arguments to the right registers.

```
        /* Define the user stack in the address space */
        result = as_define_stack(as, &stackptr);
        if (result) {
                /* p_addrspace will go away when curproc is destroyed */
                return result;
        }

        /* Warp to user mode. */
        enter_new_process(0 /*argc*/, NULL /*userspace addr of argv*/,
                          stackptr, entrypoint);
```

4.2 Extending `runprogram`

It is this last part of `runprogram` that we must extend to add argument passing functionality. Notice that we are passing `enter_new_process` an `argc` of 0 and a null pointer for the argument vector. We also do not at any point actually copy the arguments out to the new process. The reason is that `runprogram` does not have access to them.

First we need to actually have the arguments available in `runprogram`. Its caller, `cmd_progthread`, in `kern/startup/menu.c` has a pointer to the arguments and the number of arguments. We only need to change the call to `runprogram` to accept the number and array of arguments that are avaialble in `cmd_progthread`.

Programming: Modify the call to `runprogram` to accept the number of arguments and array of in-kernel arguments. Pass the program name as part of the argument array. Hint: Its definition and declaration should accept the same arguments as a regular C `main` function. Note: that the decalration of `runprogram` is in `kern/include/test.h`.

Now that we have argc and argv available in runprogram, we must pass them to the process. We are currently running in a kernel context, and the arguments are in the kernel's part of the address space. Both the arguments' and the argv vector's location has to be in userspace for the program to be able to read them. To do that, we must copy the arguments out to the userspace part of the address space.

We cannot store the arguments in the heap; the userspace allocator would have no way of knowing we did so. We can try instead putting the arguments in the stack segment. In that case, though, they will be overwritten if enough data gets written to the stack if we put them in 'before' the stack pointer . We instead have to put them in memory that userspace would never attempt to use.

The solution is to place the arguments *after* the stack pointer, decrementing the latter to make room for the arguments. We essentially use the stack segment as a bump allocator, with the stack pointer being the bump pointer. We make enough room between the stack pointer and the end of userspace to fit all the command line arguments. Since userspace will never increment the stack pointer beyond the initial value we pass to it, the arguments are safe.

As an example, suppose we want to pass the string 'abcd' to userspace. The initial stack pointer value is 0x7ffff ffff. We decrement the pointer to fit the string, together with the trailing null \0, for a final value of 0x7fff fffa. The address of the string in userspace is 0x7ffff fffa.

Programming: Implement a function argcopy_out that gets passed the stack pointer by reference and a string to be copied out, decrements the stack pointer, then uses the copyoutstr function to copy the argument to the newly bump allocated space in userspace. The function returns the userspace address of the copied string.

Right now we are copying the string out, but we are not actually providing a way to userspace to find these arguments in its address space. This is because we are copying out only the arguments themselves. Instead, we need to create argv, the argument vector. This vector is a NULL terminated array of addresses, where each entry points to one of the process' command line arguments. Each call to argcopy_out provides us with the address of the argument, so we can create argv in the kernel and copy it out.

Keep in mind that argv is an array for addresses, which are 32-bit for this architecture. MIPS thus expects them to be aligned to a 4-byte boundary. In order to ensure this, round down the stack pointer to a multiple of 4 before copying it out. Also make sure to end the array with a NULL pointer.

Programming: Allocate, fill in, copy out and free an argv array as described above. Hint: Use copyout instead of copyoutstr, because the latter stops at the first zero byte it finds. We thus cannot use it to copy out arbitrary structs.

We now have both the number of command line arguments, and an array of pointers to the arguments themselves. We only need to pass them to enter_new_process.

Programming: Modify the call to enter_new_process to pass it argc and argv.

Programming: Modify kern/startup/menu.c to remove the warning "Warning: argument passing from menu not supported"

To tests passing of arguments to the first process through runprogram, use testbin/add. The test program adds the two numbers it is passed as arguments. It should produce output like this:

```
OS/161 kernel [? for menu]: p testbin/add 3 5
Answer: 8
Operation took 0.085253080 seconds
```

5 Implementing execv

Now that runprogram is working it is quite easy to implement the execv system call. The only changes is that we need to clean up past process state, and move the arguments from the old to the new address space. Both additions are due to initiating execution from userspace instead of the kernel.

The call's userspace signature is exec(char *progname, char **argv). Search for sys_execv in the kernel we find out that the function to implement execv does not currently exist. Similar to fork. We need to create it.

Programming: Add a stub function called sys_execv to kern/syscall/proc_syscalls.c and include its declaration, where necessary.Hint: Make sure the signature is the same as above.

Since we just created the system call, it does not get called in the system call vector. We need to add it, like we added the call to `sys_fork` in Assignment 1.

Programming: Add a case in the system call vector for `sys_execv`. Hint: Pass the arguments provided in the trapframe.

The next part is quite easy: We just copy the code of `runprogram` straight into `sys_execv`! The calls are nearly identical, except for the differences mentioned above. The first one is straightforward to resolve; we must destroy the old address space before we attach the new one to the process.

Programming: Call `as_destroy` on the old address space after activating the new one.

The other difference between `sys_execv` and `runprogram` is that the command line arguments are in the old address space, not the kernel. We need to copy them over before we destroy the old address space, and then copy them to the new one using the exact same code as `runprogram`. To do this, we must allocate enough space to copy them over. For this we assume a maximum number of arguments (16), and a maximum size for each argument (128). We can create an array large enough to hold arguments of any number and size within these limits, and deallocate it after we copy everything out.

Programming: Write an `args_alloc` function that dynamically allocates one array for each possible argument, then returns a NULL allocated array (also dynamically allocated) with the addresses of the former.

Programming: Write an `args_free` function that receives an array of buffers created with `args_alloc`, frees each buffer in the array, then frees the array itself.

Programming: Write an `argcopy_in` function that accepts a dynamically allocated array of buffers, sequentially copies in command line arguments from userspace using `copyinstr`, and returns the total number of strings copied in. Hint: The array in userspace is `NULL` terminated.

Programming: Use the `args_` and `argcopy_` functions in `sys_execv` for argument passing. Do not forget to modify the call to `enter_new_process` to pass `argc` and `argv` to userspace.

We cannot test your `execv` implementation unless the system calls from Assignment 1 (fork, waitpid, exit) are implemented and working properly. Therefore, there is little point in working on execv until the Assignment 1 system calls are done.

To test your implementation of `execv`, we will try the following tests:

uw-testbin/hogparty

testbin/sty

uw-testbin/argtest

uw-testbin/argtesttest

The hogparty and sty tests are run using several different server configurations. The remaining tests, which test argument passing, are run in just one server configuration.

The test program `uw-testbin/hogparty` should produce output like this:

```
OS/161 kernel: p uw-testbin/hogparty
xxxxx
zzzzyzy
yyy
Operation took 0.453588697 seconds
OS/161 kernel: q
Shutting down.
The system is halted.
```

Similarly, the test program `argtest` without any arguments should look like this:

```
OS/161 kernel [? for menu]: p uw-testbin/argtest
argc    : 1
&tmp    : 0x7fffffb0
&i      : 0x7fffffb4
```

```
&argc  : 0x7fffffc8
&argv  : 0x7fffffcc
argv   : 0x7fffffdc

&argv[0] : 0x7fffffdc
&argv[1] : 0x7fffffe0

argv[0] : 0x7fffffe8
argv[1] : 0x0

argv[0] -> uw-testbin/argtest
argv[1] -> [NULL]
Operation took 0.155976160 seconds
```

Similarly, the test program `argtest` without many arguments should look like this:

```
OS/161 kernel [? for menu]: p uw-testbin/argtest 456 howaboutthis string and another yet ag
argc   : 8
&tmp   : 0x7fffff68
&i     : 0x7fffff6c
&argc  : 0x7fffff80
&argv  : 0x7fffff84
argv   : 0x7fffff94

&argv[0] : 0x7fffff94
&argv[1] : 0x7fffff98
&argv[2] : 0x7fffff9c
&argv[3] : 0x7fffffa0
&argv[4] : 0x7fffffa4
&argv[5] : 0x7fffffa8
&argv[6] : 0x7fffffac
&argv[7] : 0x7fffffb0
&argv[8] : 0x7fffffb4

argv[0] : 0x7fffffe8
argv[1] : 0x7fffffe4
argv[2] : 0x7fffffd8
argv[3] : 0x7fffffd0
argv[4] : 0x7fffffcc
argv[5] : 0x7fffffc4
argv[6] : 0x7fffffc0
argv[7] : 0x7fffffbc
argv[8] : 0x0

argv[0] -> uw-testbin/argtest
argv[1] -> 456
argv[2] -> howaboutthis456
argv[3] -> string
argv[4] -> and
argv[5] -> another
argv[6] -> yet
argv[7] -> ag
argv[8] -> [NULL]
Operation took 0.293392080 seconds
```

6 Expanding TLB functionality

For the TLB we will add two kinds of functionality. First, we will handle TLB exhaustion due to the program using too many pages at once. We will then turn all code or `text` segments into read-only to prevent accidentally overwriting them.

6.1 Managing TLB exhaustion

For this functionality we need to change the code in `kern/arch/mips/vm/dumbvm.c`. Let's see what the page fault handling code in `vm_fault` currently does:

```
int
vm_fault(int faulttype, vaddr_t faultaddress)
{
        ...
        switch (faulttype) {
            case VM_FAULT_READONLY:
                /* We always create pages read-write, so we can't get this */
                panic("dumbvm: got VM_FAULT_READONLY\n");
            case VM_FAULT_READ:
            case VM_FAULT_WRITE:
                break;
            default:
                return EINVAL;
        }
        ...
```

During the fault we first check what kind of fault triggered the handler. Either it was a read/write fault on a nonexistent entry, or the process tried to write to a read-only entry. The latter case leads to a panic, while the former ones are handled.

```
        ...
        if (faultaddress >= vbase1 && faultaddress < vtop1) {
                paddr = (faultaddress - vbase1) + as->as_pbase1;
        }
        else if (faultaddress >= vbase2 && faultaddress < vtop2) {
                paddr = (faultaddress - vbase2) + as->as_pbase2;
        }
        else if (faultaddress >= stackbase && faultaddress < stacktop) {
                paddr = (faultaddress - stackbase) + as->as_stackpbase;
        }
        else {
                return EFAULT;
        }
        ...
```

The code then calculates the physical address of the fault from the virtual one. This calculation happens because there is currently no page table implementation.

```
struct addrspace {
  vaddr_t as_vbase1;
  paddr_t as_pbase1;
  size_t as_npages1;
  vaddr_t as_vbase2;
  paddr_t as_pbase2;
  size_t as_npages2;
  paddr_t as_stackpbase;
};
```

Each address space is composed of three contiguous segments. One holds the code (`text`), one holds the data, and one is the stack. The segments are preallocated, so we do not lazily allocate pages on faults like in mature operating systems. Moreover, since we are using contiguous segments we only need to know what range a virtual address falls into to find the translation.

```
        ...
        /* Disable interrupts on this CPU while frobbing the TLB. */
        spl = splhigh();
        for (i=0; i<NUM_TLB; i++) {
                tlb_read(&ehi, &elo, i);
                if (elo & TLBLO_VALID) {
                        continue;
                }
                ehi = faultaddress;
                elo = paddr | TLBLO_DIRTY | TLBLO_VALID;
                DEBUG(DB_VM, "dumbvm: 0x%x -> 0x%x\n", faultaddress, paddr);
                tlb_write(ehi, elo, i);
                splx(spl);
                return 0;
        }

        kprintf("dumbvm: Ran out of TLB entries - cannot handle page fault\n");
        splx(spl);
        return EFAULT;
        ...
```

The final part of `vm_fault` constructs the new TLB entry and attempts to insert it into the TLB. The new entry is composed of the `ehi` and `elo` variables. `ehi` holds the virtual address and `elo` holds the physical address together with access bits. In OS/161 `TLBLO_VALID` means the page is readable, and `TLBLO_DIRTY` means it is writable.

The code turns off interrupts, then reads TLB slots by one. When it finds a free one, it inserts the new entry in it. If no TLB slots are available, an error is reported, ”dumbvm: Ran out of TLB entries - cannot handle page fault” and results in the caller code killing the process and calling `panic`, crashing the kernel.

We must change the code to overwrite a random slot if none are free, to avoid the kernel crash.

Programming: Modify the code, to remove the error message, and write the new `ehi` and `elo` values to a random TLB slot. Use `tlb_random` for that.

Next we implement read-only functionality. For this, we must check when inserting the TLB entry, whether it is a code entry or not. If it is a code or `text` segment, then mark it as read-only. However, we must be able to write to code pages when loading the ELF program into the address space. We saw that this happens in `runprogram` and `sys_execv`. Therefore, we must track whether the ELF executable has been loaded before marking code (`text`) pages as read-only in the TLB.

To do this we expand the `addrspace` structure with a boolean field `as_loaded`. The variable has a value of `0` when the address space is created in `as_create`, and is set to `1` in at the end of `as_complete_load`.

Programming: Add the `as_loaded` field to `struct addrspace` in `kern/include/addrspace.h`, properly initializing it in `as_create` and setting it in `as_complete_load` in `kern/arch/mips/vm/dumbvm.c`.

If we look into `runprogram` and `sys_execv`, we see that they call `load_elf`, which eventually calls `as_complete_load`. We should ensure this call invalidates the TLB, blowing away any read-write mappings for the code region we created during loading. Any TLB misses incurred during actual execution will thus never use stale read-write TLB entries for code. The code for invalidating the TLB entries is alreadyin `as_activate`, so we just need to copy it over in `as_complete_load`.

Programming: Add TLB invalidation code to `as_complete_load`.

Next, we actually have to check for the field. When resolving the physical address in the handler we can check whether it falls into the code segment. If it does, we do not set the dirty bit when creating the new entry.

Programming: Modify the `vm_fault` so that when the `faultaddress` is in the `text` or `code` segment, then check the `as_loaded` flag to make sure that `load_elf` has completed. If so mark the TLB entry read only, otherwise it is read-write (with the dirty bit set) just like all the other segments.

Finally, we must change the check at the beginning of the fault handler to exit gracefully instead of calling `panic`. We cannot return to the caller, since returning a fault to it also causes a panic. We instead directly call `sys_exit` with an error code in the event of a write fault on a read-only entry. The handler is called from the offending process' kernel context, so the `sys_exit` call destroys that process.

Programming: Modify the handler to call `sys_exit` with an `EFAULT` error on a write fault to a read only `text` or `code`.

To test for correct implementation, you can try the followng:

- `uw-testbin/vm-data1`: This is a simple test of reading and writing to the data segment. Correct output for this test should look as follows:

```
OS/161 kernel: p uw-testbin/vm-data1
SUCCEEDED
Operation took 0.796736057 seconds
```

- `uw-testbin/vm-data3`: This tests reading and writing to the data segment, and also reading from the code segment. Correct output for this test should look as follows:

  ```
  OS/161 kernel: p uw-testbin/vm-data3
  SUCCEEDED
  Operation took 1.336075017 seconds}
  ```

- `uw-testbin/romemwrite` This tests writing to the code segment, which the kernel is supposed to be disallowing. The output should look like this:

  ```
  OS/161 kernel: p uw-testbin/romemwrite
  Trying to write to the text segment
  This program should fail if the text segment is read-only.
  However, the kernel should not crash...
  Operation took 0.102239017 seconds
  OS/161 kernel: q
  Shutting down.
  ```

  Note that the kernel has prompted for another command (q), which indicates that it has not crashed.

- `uw-testbin/vm-crash2` This is similar to the previous test, but tests writing to read-only data. The output should look like this:

  ```
  OS/161 kernel: p uw-testbin/vm-crash2
  Operation took 0.075943937 seconds
  OS/161 kernel: q
  Shutting down.
  ```

7 Implementing a physical page allocator

Finally, we implement the physical page allocator. To better understand the modifications we have to do, let's see the current implementation.

```
void
vm_bootstrap(void)
{
        /* Do nothing. */
}

static
paddr_t
getppages(unsigned long npages)
{
        paddr_t addr;

        spinlock_acquire(&stealmem_lock);

        addr = ram_stealmem(npages);

        spinlock_release(&stealmem_lock);
        return addr;
}

/* Allocate/free some kernel-space virtual pages */
vaddr_t
alloc_kpages(int npages)
{
        paddr_t pa;
        pa = getppages(npages);
```

```
        if (pa==0) {
                return 0;
        }
        return PADDR_TO_KVADDR(pa);
}

void
free_kpages(vaddr_t addr)
{
        /* nothing - leak the memory. */

        (void)addr;
}

void
as_destroy(struct addrspace *as)
{
        kfree(as);
}
```

We see that right now there are multiple pieces missing:

- The `vm_bootstrap` call that supposedly sets up the allocator does nothing.
- The `getppages` call that backs `alloc_kpages` uses the boot time bump allocator through `ram_stealmem`.
- The `free_kpages` call does not do anything, since there is no allocator to free the pages back to.
- The `as_destroy` call does not free up the allocated process segments, it only frees the address space struct itself.

We will modify the code to actually free up allocated process memory when no longer needed. First we will create a `putppages` call that corresponds to `getppages` and frees physical pages back to the allocator. The call will have a signature of `void putppages(paddr_t paddr)`, so it will use physical pages like `getppages`. The virtual address passed to `free_kpages` is kernel virtual. We will create in `kern/arch/mips/include/vm.h` a call, `KVADDR_TO_PADDR` that does the opposite of the existing `PADDR_TO_KVADDR` call. We will then use it to call `putpages` from `free_kpages`. We will also free the three segments of the address space during `as_destroy` by calling `getppages` on their physical addresses.

Programming: Add a stub for `putppages`, change `free_kpages` and `as_destroy` to use it.

Now that we are done with the scaffolding we can properly implement the allocator. We will use a straightforward bitmap allocator design to keep things simple. We have three tasks:

- bootstrap the allocator,
- change `getppages` to use the allocator structure, and
- implement `putppages`.

These three functions are basically methods of the allocator data structure, so defining the latter makes implementing the former trivial.

The allocator is an array of 32-bit integers. Each entry corresponds to a physical page. We use 32-bit integers both for simplicity and to be able to hold enough information to make debugging easier. Using 32 bits is incredibly wasteful in terms of space usage (we could do with 2 bits per page), but it does not really matter in our case.

We represent allocations by filling in the corresponding pages' entries. We cannot just mark each pages as allocated or free, because when freeing we only use the start address of the allocated region. We thus need to be able to determine exactly which pages an allocation returned. This prevents us from accidentally freeing pages that were allocated contiguously, by a diffent call to `getppages`.

An easy way to distinguish between allocations is to store the size of the allocation on the first page returned. The entries of the other pages are filled with a poison value to make it trivial to test if we are freeing a valid region. Suppose we do an allocation for page `0x10000`, for example, and the allocation has size 4. The allocator entry for `0x10000` has a value of 4, and those of `0x11000`,`0x13000` and `0x13000` are filled with `ALLOC_POISON`, a poison value we define at the top of the file.

Now that we have determined how to represent allocations, we need to find a way to allocate the allocator's map. We will do this in `vm_bootstrap`, called in `boot()` after most of the rest of the kernel is set up. This call overrides the boot time bump allocator and sets up the bitmap.

To implement the allocator we first call `ram_getsize` to find out the limits of our physical addresses. Using those limits we know how large the allocator's array has to be. We will use the beginning of the available physical addresses to store the map. We thus need to translate the first available physical address to kernel virtual using `PADDR_TO_KVADDR`. We can use the resulting pointer as a pointer to a global array of integers called `physmap`.

Now that we have a map of all physical pages, we need only mark any pages already in use as inaccessible using `ALLOC_POISON`. Such pages either got allocated during boot using `ram_stealmem`, or are in use by the map itself. We know that all physical pages from 0 to the page of the first page reutnred by `ram_getsize` are thus unavailable. We also know the starting address and size of the map, and thus know which pages to reserve for it.

Programming: Implement `vm_bootstrap` by allocating space for `physmap` and initializing its space, as described above.

We now have the map, and we have described how to implement allocation and freeing. One small issue we have to take care of is making sure that `getppages` and `putppages` work at boot time. We thus add a `physmap_ready` global variable that only gets set at the end of `vm_bootstrap`. If the variable is not set, `getppages` should use `ram_stealmem` as it currently does, and `putppages` should do nothing.

Programming: Add the logic to keep using the bump allocator before the main allocator is set up.

Finally, we implement the allocation and deallocation routines. The `getppages` call iterates through the array until it finds a large enough contiguous array of pages. It then reserves them by writing the size of the allocation to the entry of the first page, then poisons the rest. The `putpages` call reads the size of the allocation, then frees the right number of pages.

Programming: Implement `getppages` and `putppages`. Hint: Make sure to use physical addresses. Use the existing `stealmem_lock` spinlock to serialize calls to the allocator.

To test for correct implementation, you can try the followng:

- `testbin/sort`: This runs the sort test program six times in sequence. The machine is configured with about 2MB of memory, and the sort program requires about 1.2MB to run. Each run of the sort program should produce output like this:

  ```
  OS/161 kernel: p testbin/sort
  testbin/sort: Passed.
  Operation took 4.804249320 seconds
  ```

  If the program does not report Passed, or if the kernel crashes before prompting for the next command, the run has failed.

- `uw-testbin/hogparty`: Run the hogparty test program five times in sequence (without quitting the kernel). The machine is configured with about 1MB of memory. hogparty is about 55KB, and it creates three hogs, each of which requires about 50KB, so that total size of the process tree is a bit over 200KB. Unlike the widefork test, this one requires execv in addition to widefork.

  Each run of the hogparty program should produce output like this:

  ```
  OS/161 kernel: p uw-testbin/hogparty
  yyyyyx
  xxxx
  zzzzz
  Operation took 0.418434697 seconds
  ```

8 Submitting Your Work

To submit your work, you must use the `cs350_submit` program in the `linux.student.cs` computing environment.

Important! You must use `cs350_submit`, not `submit`, to submit your work for CS350.

Note the usage for `cs350_submit` command is as follows

```
% usage: cs350_submit <assign_dir> <assign_num_type>
```

The `assign_dir` is the path to the root folder of the programming assignment. For the A3-kernel side programming assignment, the `assign_dir` is the path to your `os161-1.99` folder.

The `assign_num_type` for the kernel side is `ASST3`.

The argument `assign_dir` in the `cs350_submit` command, packages up your OS/161 kernel code or `userspace` program, respectively, and submits it to the course account using the regular `submit` command.

This assignment only briefly summarizes what `cs350_submit` does.

You may submit multiple times. Each submission completely replaces any previous submissions that you may have made for this assignment.