

CS169 Programming Assignment 4: VM

Assignment Out: Monday, Nov. 10, 2008
Code Handin Due: Tuesday, Dec. 9, 2008
Demo Due: Thursday, Dec. 11, 2008

1 Introduction

At this point, Weenix is a thread library and some thin wrappers around device drivers with filesystem support. By the end of this assignment, Weenix will be an operating system. With the addition of VM, your kernel will start managing user address spaces, running user-level code, and servicing system calls.

This assignment is substantial, and also very prone to difficult bugs. First of all, making sure that your kernel functions perfectly up to this point should be your first priority, although you will undoubtedly uncover more bugs in the course of this assignment. Make sure to start early and talk with your TAs frequently, as it is very easy to get lost in this assignment.

This assignment historically has been handed out as two separate assignments, but over the course of time they have blended together to the point where over the past few years they have been handed out together. The VM help session slides are the most up-to-date resource. We will work on updating the KHG. You should also look at the simulator guide for reference on some TLB functions.

Because VM Bugs can spring up in old sections of code, this is where you will probably find out whether or not your implementations of kernel code, VFS/S5FS code or drivers are busted. We'd like to point out (if you haven't been using them already), that the debug functions in `printdbg.c` are EXTREMELY useful here. Being able to print the entire address space or having pre-written code to examine a TLB entry is really nice. Just our \$0.02.

2 VM Areas and the TLB

2.1 VM Areas

In the first part of the VM assignment, you must bring up Weenix's low-level virtual memory facilities: `vm_areas`, and the functions that interface with the simulator's TLB. You need to make sure that you can create and add `vm_areas` to a process's virtual address space, and properly respond to the TLB exceptions that the simulator raises.

Functions you will need to write in `vm_area.c`:

- `void add_vm_area(proc_t *proc, vm_area_t *newvma)`
- `int find_page_range(proc_t* proc, int npages)`
- `vm_area_t* find_vm_area(proc_t* p, unsigned long vfn)`

2.2 TLB Functions

To understand the TLB, make sure you read the lectures on Virtual Memory very closely, and take a good look at section 5.4 of the Simulator manual. Some VM management is done for you, but you will have to fill the entries if they don't exist (by bringing in pages from disk), fill and flush the TLB to aid in lookups, and manage Copy-On-Write pages (which will be very important with `fork`). You will also need to make sure that pages that are not backed by a file (anonymously mapped) remain pinned, so they don't get paged out by accident.

Functions you will need to write in `tlb.c`:

- `void tlbrefill(unsigned long vaddr, int cause, sim_context_t *ctx)`
- `void tlbmodified(unsigned long vaddr, sim_context_t* ctx)`
- `void tlb_clean(void)`

3 mmap, munmap, and process memory management

3.1 mmap and munmap

These functions are the higher-level methods for creating and destroying `vm_areas`. For `mmap` and `munmap`, read the manual pages (preferably the Solaris ones)¹ to understand all the interactions between the various parameters. You are also strongly advised to revisit our version of `mman.h` here so that you'll know which flags to support and which ones not to. It would be a royal pain if you started to write and support some flag only to find out later on that you didn't need to.

Functions you will need to write in `mmap.c`:

- `int do_mmap(void* addr, int len, int prot, int flags, int fd, int off)`
- `int do_munmap(void* addr, int len)`

3.2 Process memory management

There are a few other routines you will need to have. Some of these you will probably recognize as unused stubs from Kernel I.

Functions you will need to write:

- `void proc_clear_vm(proc_t *p) (proc.c)`
- `void proc_add_anon_map(proc_t *proc, int lpage, int len, int prot) (proc.c)`
- `void *do_brk(void *addr) (brk.c)`

Functions you will need to edit:

- `kthread_t* kthread_create (kthread.c)`

Don't forget that you'll now need to change the thread creation mechanism so that it will differentiate between a user stack and a kernel stack. Up until now, everything's been running in kernel mode.

¹We're not kidding. The Sun manpages are loads better than the Linux ones. Use `sunman` to read them.

4 Traps and user memory

4.1 Accessing User Memory

The code to handle traps and access user memory has been done for you. Most of these functions need to check to see if a region of user memory is valid. This happens through the magic of the `range_perm` and `addr_perm` functions, which you must write.

Functions you will need to write in `useracc.c`:

- `int addr_perm(struct proc *p, const void *vaddr, unsigned perm)`
- `int range_perm(struct proc *p, const void *avaddr, int len, unsigned perm)`

4.2 Syscalls

The majority of the system calls have been written for you. In order to give you some understanding of the process involved, however, you will need to write a few yourself.

Functions you will need to write in `syscall.c`:

- `int sys_read(read_args_t *arg)`
- `int sys_write(write_args_t *arg)`
- `int sys_getdents(getdents_args_t *arg)`

5 Final Lap: Fun with fork and exit

You are finally ready to write the `fork()` system call. A good implementation of the previous sections is essential; `fork` is complicated enough without having to debug the rest of your VM code at the same time. To avoid 1000+ line files, we are providing the documentation for `fork` here instead of in the source file.

Functions you will need to write:

- `int do_fork(void) (fork.c)`
- `kthread_t * kthread_copy(kthread_t *old_thr, sim_context_t *sc, proc_t *target) (kthread.c)`

- `void proc_exit(proc_t *proc, int status)` (`proc.c`)

`fork` is a moderately complicated system call. We present it here as one long algorithm, but it will make your life much easier if you break it down into separate subroutines. Close attention to detail will help you; an under-debugged `fork` can cause subtle instabilities and bugs later on.

Bugs in the virtual memory portion of `fork` tend to cause bizarre behavior: user processes' memory may not be what it ought to be, so almost anything can happen. The user process may end up executing data, jumping into the middle of the wrong subroutine, etc. These sorts of bugs are very, very difficult to track down. For this reason, you should code more defensively than you may be used to. Assert everything you can, PANIC at the first sign of trouble, and include apparently unnecessary sanity checks.

Above all, be sure you really understand this algorithm before you start coding. If you try to implement it before you understand what you are trying to do, you will write buggy code. You will then forget that you've written some of that buggy code, and waste time debugging code that you should have thrown away.

Here are the steps you have to take. You don't have to use this order, but consider carefully the order in which you do these steps, particularly in case they fail. (For example, if a step fails after you've allocated some memory, you'll need to free that memory. So it makes sense not to make any undoable changes until completing all the steps which can fail (which are usually allocations).)

- Allocate a free `proc_t` structure out of `procs` using `new_proc()`.
- Copy the `vm_areas` from the parent process into the child. But you'll have to increase the ref counts on the underlying `vm_object` for non-anonymous mappings.
- For each private mapping, point the `vm_area` at a new "shadow" `vm_object`, which in turn should point to the original `vm_object` for this `vm_area`. This is how you know that the pages corresponding to this mapping are copy-on-write. Be careful with refcounts. Also note that for shared mappings, there's no need to copy the `vm_object`.
- Flush the TLB, because the parent process might still have writable TLB entries for pages we have just marked COW. If the process proceeded to write to such a page, Weenix would not notice that a write had taken place, and `fork`'s copy-on-write guarantee would be broken.
- Copy the file table of the parent into the child. For each file in the parent's file table, point the child at the same file. Increase the reference count on the underlying file, with `fref`.
- Set the child's current working directory to point to that of the parent; increase the reference count on the working directory.

- Now we have to make a copy of the current thread in the child process². Allocate a new `kthread_t`, and copy the current thread into this new thread. Set the new thread's process to be the child process. Don't forget to give the new thread its own kernel stack. Then add the new thread to the child process. You should do most of this using `kthread_copy`.
- Set the return value for the new thread to 0, using `SIM_TRAP_RETURN`.
- Set the `p_pproc` field in the child process.
- Make the new thread runnable.

You will have to revisit your implementation of `exit` from Kernel 1. Be sure that your `exit` is actually releasing resources; your OS should be able to run the following program fragment for a long time:

```
for (;;) {
    if (fork() == 0)
        exit(0);
    else
        (void) wait(0);
}
```

In a real OS (or with some optimizations to Weenix) you would be able to run this indefinitely, but because our implementation doesn't clean up shadow objects, we will run out of memory once we have a really long chain of shadow objects.

When you `exit`, use `proc_clear_vm` to free all `vm_area` structures associated with this process. (Note that you can reuse your `munmap` implementation to accomplish this).

6 Other

There are a number of other functions which you might remember seeing in earlier assignments, spread throughout the kernel, which you need to find, and either write or update. These functions are all fairly small, but if you miss one, things will break. An example might be `special_file_mmap`. While you're at it, take a look in `vm_object_an.c`.

²Since the baseline Weenix doesn't use multithreaded processes, our `fork` is the equivalent of Solaris' `fork1(2)`

7 Testing

Testing your code at this point becomes rather difficult, since you must be able to create data and text in user land and execute it. This is an order of magnitude more difficult than creating kernel-mode threads as you have in past assignments. Thankfully, most of the gory details have been written for you (see Chapter 6 of the *Kernel Hacker's Guide*).

As you have no doubt discovered if you've made it this far in the course, incremental testing is key. In VM, this is more true than ever before. Because errors can pop up anywhere and be caused by one random line somewhere completely unrelated, testing your code is crucial. By integrating your VM code with the execution and linking/loading stage, you will find out how strong your VM code is. Though we have given you a step by step breakdown of what should be done, it might be better to get the VM area and TLB code working first. Then you can write syscalls, and immediately begin testing your TLB code, since this is an area prone to difficult bugs. You can do this by starting up a process in your bootstrap code, and making a call to 'exec'. Don't worry about forking yet, just try and get the first /bin/hello to show up in one of your terminals. If you can get this step done, you are well on your way to finishing.

The best way to start is to switch your makefile in the 'cmd' directory to use the static ELF format, since this is far simpler than the dynamic ELF format. To do this, uncomment the line that says `SUFFIX = .static`. This will cause static ELF executables to be built. At this point, you should use `do_execve()` to try to execute the following commands:

- **/bin/hello** – A simple “Hello world!” test. Getting this to execute properly should be a big step in the VM assignment.
- **/bin/forkbomb** – A forkbomb test, as described in the previous section.
- **/bin/stress** – A test which stresses a few different parts of the system. It also includes a forkbomb test.
- **/bin/usertest** – A test in which you can fill in with whatever you want.

Once you get these basic tests working, you should switch your makefile back to build the dynamic ELF versions, and make sure you can still execute all of these commands. You will find the `dbg_dump_mappings()` function very useful in figuring out what is going on. If you get this far, then you are ready for the final step.

8 Final Integration

At this point, you should have functioning dynamic ELF execution, and a `fork()` call that works to the best of your knowledge. Now, you are ready to complete the Weenix system by fully integrating with the binaries that we have provided for you. All that you need to do is call `do_execve("/bin/init")` in your first process, and if you are really lucky, everything will “just work”. Once you have `/bin/init` running, you will see 3 shells started up on each tty. This shell will let you execute all of the TA-provided commands, as well as some builtin commands. In addition to the commands above, you should be able to run:

- **check all** – Runs builtin shell tests on various facilities
- **vfstest -f** – Runs the VFS tester using the s5fs filesystem instead of testfs. Note that you must modify your simconfig file to have a second disk in order for this to work.
- **s5fstest** – Runs the S5FS tester
- **args** – Echos command arguments and the current environment
- **cat** – Displays the contents of a file
- **halt** – Brings the system to a halt. This is implicitly done if you exit from each of the three shells.
- **ls** – Displays the contents of a directory
- **overflow** – This purposely generates a stack overflow. It should die with an address error, but your shell should continue executing.
- **uname (-a/-T)** – Gives system information.
- **ed** – Original 'ed' implementation, precursor to sed and vi. Only hardcore hackers need apply.

In addition to just having these commands work individually, you should be able to stress the hell out of your system. This means running `stress`, then `repeat 100 vfstest -f`, while in the other two shells, running `s5fstest 100`. There will undoubtedly be errors displayed (from `s5fstest` running concurrently), but nothing should crash. If your implementation can survive this torture test, then you are probably good to go.

9 Handing In

There are two due dates for this assignment. The first is a code handin, which you should be familiar with by this point. This will give your TA ample time to look over your code. The second

date is the demo due date. This date is absolutely inflexible (short of filing an incomplete with the registrar). Note that if you have conflicts or a tight final schedule, you can certainly hand in and demo before this date.