Homework 5

W4118 Fall 2023

UPDATED: SATURDAY 11/18/2023 at 2:10pm EST
DUE: TUESDAY 11/28/2023 at 11:59pm EST

All homework submissions are to be made via **Git**. You must submit a detailed list of references as part of your homework submission indicating clearly what sources you referenced for each homework problem. You do not need to cite the course textbooks and instructional staff. All other sources must be cited. Please edit and include this **file** in the top-level directory of your homework submission in the `main` branch of your team repo. **Be aware that commits pushed after the deadline will not be considered.** Refer to the homework policy section on the **class web site** for further details.

Programming Problems:

Group programming problems are to be done in your assigned **groups**. The Git repository for your group has been setup already on Github. It can be cloned using:

```
git clone git@github.com:W4118/f23-hmwk5-teamN.git
```

(Replace `teamN` with the name of your team, e.g. `team0`). This repository will be accessible to all members of your team, and all team members are expected to commit (local) and push (update the server) changes / contributions to the repository equally. You should become familiar with team-based shared repository Git commands such as **git-pull**, **git-merge**, **git-fetch**.

All team members should make at least *five* commits to the team's Git repository. The point is to make incremental changes and use an iterative development cycle. Follow the **Linux kernel coding style** and check your commits with the checkpatch.pl (or provided run_checkpatch.sh) script on the default path in the provided VM. Errors from the script in your submission will cause a deduction of points.

For students on Arm Mac computers (e.g. with M1 or M2 CPU): if you want your submission to be built/tested for Arm, you must create and submit a file called `.armpls` in the top-level directory of your repo; feel free to use the following one-liner:

```
cd "$(git rev-parse --show-toplevel)" && touch .armpls && git add .armpls
&& git commit -m "Arm pls"
```

You should do this first so that this file is present in any code you submit for grading.

For all programming problems you should submit your source code as well as a single README file documenting your files and code for each part. Please do NOT submit kernel images. The README should explain any way in which your solution differs from what was assigned, and any assumptions you made. You are welcome to include a test run in your README showing how your system call works.**It should also state explicitly how each group member contributed to the submission and how many hours each member spent on**

the homework. The README should be placed in the top level directory of the `main` branch of your team repo (on the same level as the linux/ and user/ directories).

Assignment Overview:

A core feature of memory management in Linux (and most other operating systems) is the concept of virtual address spaces. To each process running in userspace, it appears as if it is the only process using the memory. It can allocate, read, write, and deallocate memory at any allowed 'virtual' address without worrying if some other process happens to be using that same address. However, in reality multiple virtual address spaces are coexisting in physical memory.

The Linux kernel is responsible for providing this abstraction to userspace processes; it keeps track of every virtual address for each running process, and maps those addresses to physical memory. However, storing an exact, one-to-one mapping from every virtual address of every process to a corresponding physical address is clearly impossible given physical memory limitations. Instead, the kernel uses some smart data structures (in combination with hardware support) to provide the virtual memory abstraction transparently to processes, while in reality storing only a tiny fraction of all possible mappings.

In this assignment, you will explore the two main data structures the kernel uses to do memory mapping: virtual memory areas and page tables.

Virtual memory areas (VMAs), also called memory regions, are how the kernel tracks the memory that a process thinks it has allocated, along with some associated state. They represent regions in the virtual address space, but are independent of physical pages in memory.

Page tables, on the other hand, map those virtual memory addresses to physical memory addresses once real memory is actually required. In addition to mappings, the page tables also include state information (like whether pages are read-only or dirty, for example).

In this assignment, you will see how VMAs and page tables are managed and updated by tracing these updates for a specified process. We will create system calls that allow an 'inspector' process to access a continuously updating 'shadow' page table describing a small segment of the virtual address space of some other target process.

For simplicity, our shadow page table will only have a single level, and will store information about both the real page tables and the virtual memory areas of the target process. Think of the shadow page table as a single-level map from a page in the virtual address space to a frame of physical memory.

PART 1: Creating the shadow page table

1. **Definition of the Shadow Page Table**

   First, set up the data structures for the shadow page table that you will provide to inspecting processes. Since both your internal kernel logic and the

userspace processes that use your system call need to know how these are structured, you should place them in a uAPI header. Create the file `include/uapi/linux/shadowpt.h`, and add the following components:

- The `shadow_pte struct`, which represents a single entry in the shadow page table. You should store one for each page in the target virtual address range. Each entry includes a (page-aligned) virtual address at the start of the virtual page, the physical frame number (pfn) of the mapped physical page (or 0 if it is unmapped), and flags to describe the state of the entry, which we refer to as its mapping state.

```
/* The data structure representing a single entry in the page ta
struct shadow_pte {
    unsigned long vaddr; /* Virtual address of the base of this
    unsigned long pfn; /* Physical frame number of the assocate
    unsigned long state_flags; /* Flags describing the mapping s
};
```

- Some constant flag values that will be packed in the state_flags field.

```
#define SPTE_VADDR_ALLOCATED 1    /* the page is allocated */
#define SPTE_VADDR_ANONYMOUS 2    /* the page is for anonymous r
#define SPTE_VADDR_WRITEABLE 4    /* the page is writable (else

#define SPTE_PTE_MAPPED 8         /* a physical frame of memory
#define SPTE_PTE_WRITEABLE 16     /* the frame of memory is writ
```

- A small number of helper macros to read whether a particular state flag in a shadow_pte is set.

```
/* Macros for testing whether each state flag is set */
#define is_vaddr_allocated(state) (state & SPTE_VADDR_ALLOCATED)
#define is_vaddr_anonymous(state) (state & SPTE_VADDR_ANONYMOUS)
#define is_vaddr_writeable(state) (state & SPTE_VADDR_WRITEABLE)

#define is_pte_mapped(state) (state & SPTE_PTE_MAPPED)
#define is_pte_writeable(state) (state & SPTE_PTE_WRITEABLE)
```

- A `user_shadow_pt` struct to represent the full shadow page table for the target range, which just includes a start and end virtual address, and an array of all the shadow page table entries in that range:

```
#define MAX_SPT_RANGE (8388608)

/* The data structure representing the full page table */
struct user_shadow_pt {
unsigned long start_vaddr; /* first virtual address in the targe
unsigned long end_vaddr; /* last virtual address in the target
struct shadow_pte __user *entries; /* array of all page table en
```

```
};
```

Note that an earlier version of the homework specification indicated that `end_vaddr` should be the last virtual address in the target range instead of the one after it, but this was inconsistent with some other aspects of the specification. We will accept solutions based on either interpretation of `end_vaddr`.

- Note that because you are going to pass this structure from userspace to the kernel with a userspace address, we mark that address with the `__user` macro.
- It's also a good idea to add a reasonable limit to the maximum range (i.e. end_vaddr - start_vaddr) a shadow page table can contain, since this implementation isn't particularly space efficient.

As you write your code, you may find it helpful to add some helper functions or macros to this header file to more easily access information related to these structures. However, do not modify the provided data structures or add additional data structures to pass information between userspace and the kernel. The provided data structures make up the minimal and complete API for your system call; any additions you make should be for convenience only.

In addition to the UAPI header, you should create a separate `include/linux/shadowpt.h` header. Here, you'll want to both include the UAPI header (with `#include <uapi/linux/shadowpt.h>`) and define any additional data structures, short helper functions, or prototypes that might need to be accessible from within the kernel, but shouldn't be exposed to userspace. When you do this, make sure you have appropriate include guards in both header files.

Tasks:

1. Add a UAPI header including the provided data structures (with the **exact** provided definitions) and any other necessary information to be used in the shadow page table in a header file.
2. Add an internal header including any data structures/prototypes you will use to internally manage the shadow page table.

Notes:

1. To make your UAPI header accessible in userspace, don't forget to run `make headers_install INSTALL_HDR_PATH=/usr` after compiling and booting into your new kernel. This command will install the headers found under `include/uapi/` in your Linux source tree into `/usr/include/`. Now you should be able to `#include <linux/shadowpt.h>` from userspace. Make sure that you are able to run `make headers_install` with no compilation errors. We will consider an error when running `make headers_install` as your kernel failing to compile.
2. If you're curious about the motivation behind separating UAPI and internal kernel headers, check out **this article** explaining why the separation was introduced.

2. **System call to create the shadow page table**

Next you will write the system call to create our shadow page table and make it available to the inspector process, but don't worry about populating or destroying it yet.

The first system call (with syscall number 451), should have the following interface:

```
/*
 * Syscall No. 451
 * Set up the shadow page table in kernelspace,
 * and remaps the memory for that pagetable into the indicated u
 * target should be a process, not a thread.
 */
long shadowpt_enable(pid_t target, struct user_shadow_pt *dest)
```

System Call Usage/Behavior

An inspector process will call this system call with the pid of the process it would like to inspect and a `user_shadow_pt` struct. This struct should be populated to contain the target start and end addresses, and a linear, preallocated entries array.

In the system call function, the kernel should take these inputs, validate them, and build a corresponding shadow page table in kernel space. Once this has been done, your function should remap the pages storing the shadow page table into the inspector's virtual memory, at the entries pointer provided in the `user_shadow_pt` struct. Once this is done, the entries pointer provided by the userspace process should map to the same physical memory as the shadow page table created by the kernel.

Input Checking

If any of the provided arguments are invalid (for example, if the target process does not exist, or the start address is greater than the end address), your system call should return `-EINVAL`. In addition to basic sanity checking of the inputs, there are some additional limitations you should impose on callers:

- Only one process can be inspecting another at a time. If a process calls the system call while another process is using it, they should receive `EBUSY`.
- Only a superuser should be able to make the system call, otherwise, it should give `EPERM`.
- The start address and end address should be page aligned. If they are not, the system call should shift them down to the nearest aligned values.
- The target range should not span more than `MAX_SPT_RANGE` virtual addresses. If it does, the system call should reduce the range to be `MAX_SPT_RANGE`.
- If either of the previous two happen, an updated `user_shadow_pt` struct with the new range values should be copied back to the userspace pointer.

Remapping Details

Unlike in Homework 3, where you wrote to a ring buffer located in kernel space and later copied to userspace, we now want `shadowpt_enable()` to make the shadow page table directly accessible in the userspace process's memory. Remember that writing directly to userspace memory wasn't allowed; to safely share data with a caller, we needed to use the `copy_to_user()` function, which was (relatively) slow and could sleep.

To get around the problem of needing to repeatedly copy to userspace whenever an update is made to the page table, this time you should allocate pages in the kernel, then modify the inspector's page tables to directly point to those pages in the kernel. In other words, you will end up with two references to the same set of physical pages: one in kernel memory, which you use to write, and one in userspace memory, which you use to read. This means that any updates to the shadow page tables in the kernel effectively happen instantly for the inspecting process as well.

Take a look at the `remap_pfn_range()` function to implement this remapping. It is well documented for use in kernel drivers, but it can also be used elsewhere. Look at how it is implemented and try to understand in broad terms how it works (this will be useful for the next parts of the assignment).

Notes:

- Make sure your system call handles race conditions between multiple callers and in general handles concurrency gracefully. Under any conditions, when any number of callers make the system call, at most one should return successfully and be unaffected by the others.
- Any errors or failures in your handler should be handled appropriately. On failure, anything allocated or referenced should be cleaned up and the appropriate errno should be returned.
- **There are a lot of ways to allocate memory in the kernel.** Think carefully when choosing one for allocating the pages that you will eventually remap to userspace. (Hint: think about why `vmalloc()` would be incorrect in this scenario).
- It may seem a bit weird that we allocate memory for the page table twice (once in userspace and once in kernel space). This is correct though, because what we are really doing with `remap_pfn_range()` is updating the page table entries of the inspector so that they point to the same physical pages that we allocated in kernel space.

Tasks:

1. Implement `shadowpt_enable()`, but populate the shadow table with dummy values for now.
2. Test your system call to see that you can perform remapping, and that when you make some writes in the kernel they appear in userspace.
3. Modify any relevant files in the kernel to reflect the newly added system call.
4. In addition to building a shadow page table, this system call will likely need to store some information globally about the shadow page table, so that later on we can write to it from various locations in the kernel.

3. **Destroying the shadow page table**

   After successfully remapping a shadow page table placeholder, you now should now allow disabling of the shadow page tables, both intentionally and automatically when something goes wrong.

   The system call to handle disabling should have syscall number 452, and should use the following prototype:

```
/*
 * Syscall No. 452
 * Release the shadow page table and clean up.
 */
long shadowpt_disable();
```

   System Call Usage/Behavior

   This system call attempts to disable and clean up a shadow page table. It should reset any global state set up by `shadowpt_enable()`, to allow new calls to `shadowpt_enable()` to succeed.

   Only the exact process which successfully called `shadowpt_enable()` should be able to successfully disable its shadow page table.

   Since you don't want to leave the inspector process with access to kernel pages, you should 'undo' the `remap_pfn_range()` to disconnect the virtual addresses of the process from these physical pages. There are multiple ways to do this, but a good place to begin is by looking through the `munmap()` system call to see how it breaks down a regular mmap'd region. You may also find `zap_vma_ptes()` helpful.

   Once the disable syscall is made, the entries range can either no longer be mapped (i.e. an access causes a segmentation fault in userspace) or can behave as if shadowpt_enable were never called, and the region was just mmap'd.

   Inspector Process Exit

   An important edge case you need to handle is if the inspector process exits before making the disabled system call, particularly since we only allow the inspector process itself to successfully make that call.

   You should make sure that no matter how the inspector process exits, your kernel behaves as if the disable system call was made and cleans up properly.

   Notes:

   - As with before, consider concurrency bugs. Make sure that your program behaves correctly for *any* set of enable/disable calls made at any time.
   - Think carefully about how you detect the inspector process exiting. You should make sure that you catch any exit for any reason.

Tasks:

1. Implement and test the `shadowpt_disable()` call.
2. Modify any relevant files in the kernel to reflect the newly added system call.
3. Implement handling for cleaning up when the inspector process exits.

PART 2: Updating the Shadow Page Table

Now that you can create and destroy the page table, the next step is to populate it with the correct values, both on initialization and then whenever those values change as described below. In particular, we suggest treating initialization similar to any other update to the shadow page table as the required functionality is similar. The difference is that initialization will require populating the entire shadow page table with the existing status of the VMAs and real page table by modifying your earlier implementation of `shadowpt_enable()` accordingly.

First, focus on correctly tracking changes to VMAs (both existence and state) within our target range. Then extend to tracking PTE states.

1. When Virtual Addresses are Updated

When the virtual address space representation (i.e. the VMA structs) is updated for an address (or some range of addresses) within the given range, you should update the shadow page table in kernel space for that address.

An example 'hook' function prototype might look something like this:

```
/*
 * Update the shadow page table when virtual addresses in the g
 * modified.
 *
 * This function should log properties (file-backed, anonymous,
 * writeable) for virtual addresses.
 */
void update_shadow_vaddr(struct mm_struct *mm, struct vm_area_s
```

You should track the following pieces of information about each page in the target range, and update its shadow_pte struct accordingly using the constant flag values we defined above:

1. Set `SPTE_VADDR_ALLOCATED` in the state_flags field when the address is allocated (so, immediately after an mmap or malloc call) and unset the bit when it is deallocated (so, after a call to free).
2. Set `SPTE_VADDR_ANONYMOUS` in the state_flags field when the address is anonymous, and unset the bit when it is not (i.e. for file-backed mapping)
3. Set `SPTE_VADDR_WRITEABLE` in the state_flags field when the address can be written to, and unset the bit when it is read-only.

There are a reasonably large number of edge cases that can cause an update to the VMA, so for our purposes you only need to place hooks in various places throughout the memory management code to track changes that happen for one of the following reasons:

1. Full initialization to existing VMA status when `shadowpt_enable()` is called.
2. Allocation by `malloc()` and `mmap()`.
3. Deallocation by `free()` and `munmap()`.
4. Protection change by the `mprotect()` system call.
5. Replacement by the `execve()` system call.
6. Expansion for a VMA marked `VM_GROWSDOWN/VM_GROWSUP`.
7. Full deallocation due to process exit.

Notes:

- When reading from components of the memory management system, remember that you're not the only one using them. Make sure that you grab the appropriate locks and increment reference counts as necessary while accessing shared kernel data structures.
- Note that the `SPTE_VADDR_WRITEABLE` flag is entirely separate from and is not the same as `SPTE_PTE_WRITEABLE` flag; the first relates to a VMA struct, and the second to a PTE.
- Spend some time looking through the kernel code to understand when and how virtual memory areas (VMAs) are allocated and deallocated, and when and how their state changes.
- VMAs that are marked `VM_GROWSDOWN/VM_GROWSUP` can be increased by the kernel without an explicit `mmap()`. For example, the VMA for the stack is marked either `VM_GROWSDOWN` or `VM_GROWSUP` so that its start address can continue to change as the stack size increases without repeatedly making a system call.
- When the target process exits, you should not disable the page table, nor interfere with that process' exit. Instead, you should mark all the virtual pages in your target range as deallocated (and stay that way until disable is called or the inspector process exits).

Tasks:

1. Implement `update_shadow_vaddr()`, or some similar update function
2. Add `update_shadow_vaddr()` calls in various kernel locations to track the desired changes.

2. **When Physical Pages are Updated**

Just like VMA structs, page table entries (PTEs) have some state flags associated with them, and can either be mapped or not. Your shadow page table should track the current state of PTEs in addition to VMA flags for particular pages.

Again, it makes sense to create some kind of update function that you can call to track changes. An example could look something like this:

```
/*
 * Update the shadow page table when a physical page is updated
 */
void update_shadowpte(struct mm_struct *mm, unsigned long vaddr
```

You should track the following pieces of information for each virtual page in the target range:

1. Set `SPTE_PTE_MAPPED` when the virtual page is associated with a physical page (via a new PTE). Unset the bit when the mapping is removed.
2. When `SPTE_PTE_MAPPED` is set, save the physical frame number of the associated page in the 'pfn' field of the shadow_pte struct.
3. Set `SPTE_PTE_WRITEABLE` when the PTE is marked as allowing writes, and unset the bit if the PTE is read only.

The key to this part is identifying where in the kernel a PTE is modified. Again there are a few edge cases you don't need to handle, but you should track:

1. Full initialization to existing PTE status when `shadowpt_enable()` is called.
2. Any time a new PTE is inserted into the page table (for example, this could happen when a user process tries to write to a particular address for the first time).
3. Any time a PTE's write protection changes as a result of a page fault.
4. Any time a PTE's write protection changes as a result of the fork() or execve() system calls.
5. Any time a PTE is invalidated, or marked to be removed from the page table (see the MMU notifier hint below for suggestions on how to implement this).

For simplicity, you do not need to track updates that involve huge pages, but your shadow page table should work correctly to track standard 4 KB pages even in the presence of huge pages.

Notes:

- Your code will require you to "walk the page tables" at some point. While there are many functions that already exist in the kernel to do this, they are pretty much all far too complicated for our purposes. You should implement your own function for walking the page table rather than trying to use an existing function. You may find it helpful to review kernel functions such as `__handle_mm_fault`, `__do_page_fault`, and `find_vma`, noting in the latter case the difference between that function's behavior and `vma_lookup`.

- You might want more than just the provided example update_shadowpte() function to behave in different contexts (with different locks held, for example).
- A nonexistent PTE can be detected either if the pte_none(pte) function returns true, or if some containing table higher in the page tables does not exist. Note that this is different from the PTE_PRESENT bit, which indicates whether the page has been swapped to disk (and which we aren't asking you to track).
- Once again, make sure you keep concurrency in mind. Many processes all must access the kernel's memory related data structures at once, so be sure to grab the appropriate locks and references when you read and write kernel shared data.
- Be sure to understand how the kernel enforces copy-on-write protections when a process forks–it will help you complete the assignment, and is an excellent demonstration of the efficiency gains the kernel makes using smart handling of memory.
- There are a huge number of places in the kernel where PTEs are invalidated, and placing an update after each one would be extremely time consuming. Luckily, we're not the only ones interested in page tables, and as a result the kernel provides the **MMU notifier**, which allows us to register our own callback functions. These callback functions will be called whenever a particular range of PTEs is going to be invalidated.
  - **Read this article** for more information about when the MMU notifier was first implemented, and **this one** for an update about recent changes that could provide some context for how to use it.
  - For an example of a kernel component that uses the MMU notifier, check out KVM, the hypervisor built into Linux.
  - You will need to register and unregister for callbacks with the notifier when you enable and disable your shadow page table. Think carefully about concurrency implications of each of these calls, and about where you place them in relation to the rest of your code.

Tasks

1. Implement `update_shadowpte()` to update your shadow page table.
2. Implement your own function for walking the page table.
3. Place calls to `update_shadowpte()` in as many locations as required to cover the mentioned scenarios.
4. Handle the scenario where the process you are tracing exits.

PART 3: Getting contents from a physical address

With the previous system calls, you now should be able to inspect a target process and see its virtual to physical mappings for some range in userspace. To help test this functionality, you should implement one final system call with the following prototype:

```
/*
 * Retrieve the contents of memory at a particular physical page
 */
long shadowpt_page_contents(unsigned long pfn, void __user *buffer
```

This system call should take in a physical address and a userspace buffer. If the caller has superuser permissions, the shadow page table is active, and the provided pfn is currently mapped by a target process virtual address in the target range, then the system call should copy the contents of that page into the userspace buffer.

Notes:

- Make sure you do proper input and permission checking, and return relevant errno values in failure cases.

Tasks

1. Implement `shadowpt_page_contents()` to get the contents from a physical page.

PART 4: Test your page table

Write a user program that calls your system call and reports changes to the page table information. The program should be in the `user/part4` directory of your team repo, and your Makefile should generate an executable named **inspector**. The program should be run as follows:

```
$ ./inspector <target_pid> <start_address> <end_address>
```

You should properly initialize your `user_shadow_pt`, which you will be passing as an argument to `shadowpte_enable()`, and allocate memory for the `struct shadow_pte` array. This memory is what will be eventually remapped in the syscall itself. Once you enable the page table, you should repeatedly print out the contents of your shadow page table once per second until the user terminates the program by pressing ctrl+c. Make sure you set up a signal handler so that the `shadowpt_disable()` system call is still properly called when you exit this way.

In addition, you should also write another program, which compiles to the executable **target**. This program will generate a variety of page table changes for you to observe with the inspector program. This program should update the contents of the page table by writing to and unmapping pages within its own virtual address space. You should be able to see the page table updating in the inspector as you perform these writes/unmaps.

Again, the only changes you need to generate are the ones you were asked to track above.

Notes:

- It's a bad idea to use malloc to prepare a memory section for mapping page table entries, because malloc cannot allocate memory with more than `M_MMAP_THRESHOLD` (128KB by default). Instead you should consider using the mmap system call; take a look at `do_mmap()` in mm/mmap.c to set up the proper flags to pass to mmap.
- Once you've mmap'ed a memory area for remapping page table entries, the kernel will create a Virtual Memory Area for it. The flags for the memory area should include the `MAP_ANONYMOUS` and `MAP_PRIVATE` flags.

Tasks

1. Implement the **inspector** and **target** executables to test your page table.

PART 5: Written Questions

Write answers to the following questions in the `user/part5.txt` text file, following the provided template **exactly**. Make sure to include any references you use in your `references.txt` file, and that you are referencing the correct kernel version in bootlin (**v6.1.11**). For reference, the URLs you answer with should be in the following format:
https://elixir.bootlin.com/linux/v6.1.11/source/kernel/sched/core.c#L6432

1. Suppose we have a 32-bit system whose hardware only supports two levels of paging and the page size is 4KB. All levels of the page table for process 1 are allocated in physical memory, and they are stored contiguously starting at physical frame number (pfn) 10. In other words, the pgd is stored starting in pfn 10, and the ptes are stored starting in the physical frame of memory iimmediately after those used to store the pgd. For process 1, if virtual address 0x3c0ffff maps to physical address 0x7d0fff, what pfn will be stored in the pgd and pte entries of process 1's page table that are used to translate this virtual address to its physical address? Write your answer using decimal numbers (not binary or hexidecimal).
2. Suppose you have a system that uses a TLB and a 4-level page table, and a virtually address cache. The TLB access time is 10 ns, the virtually addressed cache access time is 40 ns, and RAM access time is 100 ns. The TLB has a 95% hit rate, and the virtually addressed cache has a 90% hit rate. What is the average access time?
3. For a typical modern 64-bit Linux system which uses a 48-bit virtual address space for processes and 4KB sized pages, if a single frame of physical memory is mapped to a process's page table, what is the minimum amount of memory that would be required to store the process's page table when using a one-level page table, two-level page table, three-level page table, and four-level page table?
4. Specify the URL in bootlin indicating the file and line number at which the physical address is computed from the pgd, and the file and line number at which that address is loaded to the page table base register for the user space address space on a context switch, for both arm64 and x86.
5. Consider the following C program, run on a single CPU on x86 architecture:

```
int main() {
        int *ptr = NULL;
        *ptr = 5;
}
```

Identify the kernel functions from those listed below that will get executed, and put them in the order in which they will be called. Start your trace at the time when the process begins executing the first instruction of main(), and end your trace when the process will no longer run anymore. Functions may be called multiple times. Not all functions need to be used. Also, not all functions that are executed are listed below – limit your answer to include only these functions. In your answer, you should write each function **exactly how it appears below** (no extra tabs, bullets, spaces, uppercase letters, etc.), with each function on a separate line. We will be grading your answers based on a diff – if you do not follow the formatting specifications, you will not receive credit.

- arch_do_signal_or_restart
- bad_area
- __bad_area_nosemaphore
- context_switch
- deactivate_task
- dequeue_task
- do_exit
- do_fault
- do_group_exit
- do_kern_addr_fault
- do_user_addr_fault
- exc_page_fault
- exit_to_user_mode_loop
- exit_to_user_mode_prepare
- find_vma
- force_sig_fault
- force_sig_fault_to_task
- get_signal
- handle_mm_fault
- handle_page_fault
- handle_pte_fault
- handle_signal

- irqentry_exit
- irqentry_exit_to_user_mode
- kernelmode_fixup_or_oops
- lock_mm_and_find_vma
- page_fault_oops
- pgtable_bad
- __schedule
- send_signal_locked
- sigaddset
- sys_call_table
- vma->vm_ops->fault

Submission Checklist

Include the following in your main branch. Only include source code (ie *.c,*.h) and text files, do **not** include compiled objects.

- `.armpls` file for teams with M1/M2 CPUs
- `README` file
- `references.txt` file
- Implementation of system calls and additional functions added to `linux/kernel/shadowpt.c`
- Implementation of your inspector and target programs in `user/part4`
- Answers to written questions in `user/part5.txt`