User Level Page Faults

Qingyang Li

CMU-CS-20-124
August

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Seth Goldstein, Chair
Someone else

Submitted in partial fulfillment of the requirements
for the degree of Masters in Computer Science.

Keywords: User level interrupt, TLB, page table, MMU, page fault

For my grandparents:
I am resting in the shade of trees that you planted.

Abstract

Memory management has always been a responsibility shared by the hardware and the operating system. The MMU (memory management unit) walks the kernel-managed page tables to ensure safety and isolation between process' data. While users can alter some aspects of the memory system they control (e.g., permissions) these operations come with a high overhead.

We propose User Level Page Tables as a hardware/software mechanism that offers a low overhead mechanism for users to manage their application memory permissions while still having hardware enforcing those permissions. Our mechanism allows users to modify page permissions with a single write in user space and without changing privilege levels. Users can also handle many types of page faults using handler functions that they install, without crossing into kernel space.

To realize this mechanism, we modify the hardware address translation pipeline, specifically the MMU and TLB components. We also modify the Linux kernel by adding a set of "shadow" user-level page tables.

We evaluate our approach of User Level Page Tables by looking at the overhead of watchpoints. We evaluate our system using Gem5, a full system emulator, modified to implement User-Level Interrupts and User-Level Page Faults as an extension of the x86_64 architecture.

Our microbenchmarks show that even with thousands of watchpoints, we incur a slowdown of 3.25x. With the SPEC benchmarks, watching every dynamically allocated memory region, we see slowdowns range from 2.1x-15.8x. The latter slowdown suggests we need to improve our method of tracking active watchpoints.

Acknowledgments

Prof. Goldstein - Thank you for advising me these past years. I've evolved so much as a systems person under your guidance. I still have a long way to go but I now have the confidence to venture on my own. Thank you for your patience in allowing me to make mistakes. Thank you for pushing me when I was hesitant. Thank you for holding me accountable. Thank you for working on me.

Prof. Eckhardt - Thank you for providing me with resources, physical and emotional, even when I did not know I needed them. Thank you for helping me conquer my self-doubt and raise my self-esteem. I am going to miss our chats in your office.

Clem Cole, Intel Corp. - Thank you for adopting me that summer and convincing me that systems is a fun beast to play with.

Chrisma Pahka - Your patience with me is none other than god-tier. Thank you for humoring my design musings, debugging my Gem5 and compiler issues, and grounding my conceptual questions. Good luck and enjoy the rest of your PhD.

Jinghan Wang - Thank you for shouldering the hardware end of this project during the initial stages, when I barely knew what was happening. Thank you for stepping up to debug when I asked for your help.

Thank you to my roommates Caroline and Claire, for always cheering me up with some memes after a rough debugging session.

Thank you to Olivia, Alex, and Neil, for when I needed to gripe about general frustrations.

Thank you to Katie, Liam, and Doria, for getting me to play games and distracting me when I was in a rut.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

There is forever a tradeoff between the operating system's responsibilities and user capabilities. While the operating system does not strive to limit user applications, its need to ensure safety and fairness between multiple running processes makes it necessary to take away some functionality from the user. A prime example of this divide is the memory management system.

1.1 Memory Management

The operating system manages resources for multiple processes running on the same machine. Part of its responsibility is maintaining isolation between process memories; no process should be able to manipulate memory that is not its own. From each program's perspective, it is the only process running on the machine, thus it should have access to all resources, including the entire memory space. This conflict is resolved by using virtual addresses.

With virtual addressing, users cannot access physical memory directly but are still capable of allocating as much memory as they desire. The operating system maintains a mapping within kernel space between user-accessible virtual addresses and hardware-understandable physical addresses in the page table. This data structure is exclusively managed by the operating system; only certain capabilities are exposed to the user through system calls. Furthermore, this mapping is enforced by specialized memory translation hardware, so a write to a stray pointer or one that violates set permissions will cause the application to crash. Virtual addresses solve both the user's and system's needs; user applications can access the entire memory space through virtual address and cannot manipulate another process' memory for it cannot access the other process' page tables.

A typically memory access with resemble the following during execution: a virtual memory address will be held in some hardware register. The memory management unit (MMU) within the CPU will translate the virtual memory address to a physical memory address, since hardware is indexed by physical addresses. First the MMU will check the translation look-aside buffer, more commonly known as the TLB cache, to see whether the address has been translated recently before. If it has, the physical address would have been stored in the TLB and the physical page frame number will load into the output register. If it is not within the TLB, the MMU will use the virtual address to index the page tables to find the corresponding virtual address to the physical

address.

During address translation, one of several events may happen. The first, and best, case is that the physical page frame number is within the last level of the page table, and the page is mapped in memory. The MMU will return the entry's physical address, acquire the data, and execution will proceed. All other cases, signifying that some went wrong during translation, will throw a page fault exception and jump into the handler in kernel space. Under certain circumstances, the fault handler is able to recover and execution can proceed. One such case is when the physical frame number is in the table but the page is not mapped in memory, or "swapped out." This occurs when RAM cannot hold all memory pages for all running processes and must defer to storing some on a secondary storage drive, like a disk or an SSD. These pages must be brought from storage to memory and so the program can access them. Under other faulting causes, like having an absent page table entry or violating memory permissions will lead to a segmentation fault.

1.2 Interrupts and Signals

Intel categorizes these interruptions to normal execution into two groups: asynchronous and synchronous. Asynchronous interruptions, more typically know as interrupts, are thrown by hardware drivers at some input. For example, the keyboard handler will raise an interrupt when a user types a certain key. These are unpredictable as the time and execution state of user input varies from run to run. Synchronous interruptions, also know as exceptions, are thrown by the CPU when something goes wrong with instruction execution. These are predictable as the CPU will always throw the same exception when executing the faulting instruction with the given state. Page faults are an example of such exceptions. For the context of this thesis, we loosely classify page faults as "interrupts" for it interrupts normal execution. We acknowledge that the proper terminology is "exception." Every interrupt, synchronous or asynchronous, identifies with a interrupt vector. When the operating system is first booted, it will create a table saving all interrupt vectors with a corresponding function to run when the interrupt is raised; these functions are usually part of kernel code. When an interrupt is raised by hardware, the CPU will automatically save the current application execution context and execute the registered interrupt function. Once the function completes execution, the previous context will be restored and the application will commence.

The aforementioned pipeline is great for when the interrupt is intended to be handled by the operating system. However, there are several applications for when select interrupts are meant to be handled by the user code. Current systems have a method for the user to install a signal handler to be executed upon some event. In the case of a page fault, the operating system will deliver a `SIGSEGV` to the process and check for a registered user signal handler. If there is no user handler, the default kernel handler will kill the process.

Even if we overlook that invoking a user signal handler must be relayed through kernel code, the end-to-end pipeline is involves many priveledge level switches between user space and kernel space. The current path of user-level signal handling involves the application falling into kernel space code to set up the user handler trampoline, jumping to the user space handler, falling back to the kernel to restore the state pre-interrupt, and finally, resuming user program execution [Fig.

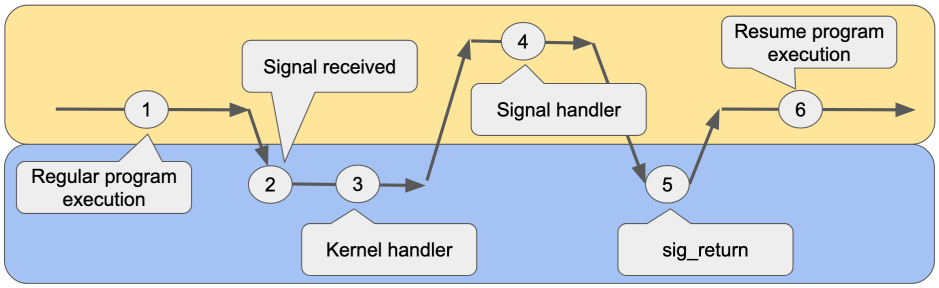

Figure 1.1: Signal Handling Pipeline

1.1].

In this thesis, we seek to imitate this user signal handler behavior by completely bypassing any kernel interference. We will implement a page faulting mechanism that jumps directly to the user handler in user space. We use this mechanism for handling user-set memory protection violations and utilize it in a watchpoints application.

1.3 Roadmap

This thesis is has the following structure. Chapter 2 will provide a more general background on previous work on faulting to user space. Chapter 3 will explain, in detail, changes made to the hardware and software components. Chapter 4 will cover changes made to adapt our mechanism for a watchpoints application along with results from benchmarks. Chapter 5 will cover viable places for improvement that we did not have a chance to explore. Chapter 6 will mention some issues for future work to address. Chapter 7 will conclude our report and provide a general reflection and discussion about this mechanism. Finally, the appendix contains bits that future developers may find useful in replicating or continuing our work.

Chapter 2

User-Level Interrupts

2.1 Problem Statement

In this project, we hope to design a pipeline in which the CPU, upon some condition, raises a special interrupt that would execute a user-registered handler function and proceed with the rest of program execution. Our goal is to completely bypass any kernel interference and thus reducing the overhead of the execution jumping from the kernel space exception handler, to the user space handler, back to the kernel space handler, and finally to user space to resume the program. Since it was triggered by the user space program, we want to stay in user space to handle the interrupt and proceed with execution. This will ideally reduce the overhead of cache misses caused from jumping back and forth between user space and kernel space.

There are some existing frameworks that we build upon, mainly the existing interrupt pipeline. There are already x86 instructions, `INT` and `IRET`, that can invoke the interrupt pipeline and return from it to resume program execution. However, the detailed microcode implementation does not entirely suit our needs in terms of saving and restoring registers. Also, the CPU is currently programmed to reference the interrupt descriptor table (IDT) to determine function to execute when it raises an exception. While we initially considered just adding another entry in the table, we found that the table was set to read-only after the initial setup; it was designed to be set up once and not modified again. However, the existing pipeline mostly suit our needs, the saving of control registers and jumping to some instruction, so we need to create another channel to invoke it.

2.2 Previous Work

There has been some previous work surrounding user level interrupt, predominantly surrounding I/O devices. These are one use case that a hardware interrupt will ultimately be handled by user-level code. Context switching between user and kernel space incurs additional cache evictions for any application whose working set exceeds the hardware cache size [4]. These cache evictions compound to a high overhead for context switching between privilege levels that do not need to happen.

Caulfield et. al. extended their Moneta storage architecture [1] with Moneta-D [2], which

provides a channel for transparently bypassing the OS for I/O operations. Their system design consists of a user library, like `libc`, mapping device memory to directly to its address space, creating a *virtual channel* that the Moneta device can directly write to. The user library will also need to register the file descriptor and its offset with the operating system along with its allowed permissions. The hardware will then use this table to enforce permissions and write to the mapped memory. Like our design, we see a division of responsibility between software and hardware components for realizing this capability. The software initializes and maintains data structure for managing memory regions, files, and their corresponding permissions. The hardware enforces the permissions and performed user requests. This differs from our design in that this interface is completely transparent to applications; it is enacted through a user library. Our design can be directly invoked by the user program.

In a design very similar to ours, the Mondrian Memory Protection (MMP) [7] scheme also strives to provide fine-grain memory protection. MMP provides memory permissions in user space with a table in kernel space that contains vectors describing permissions set to sections of data. Every memory access will have the address checked against its permission in the table through hardware. One design for this table uses the virtual address similar to that of traditional page table walking and uses sections of the address to index into a multi-level map. They also added a TLB-like structure to cache permission vector entries. MMP contains many components that are similar to our ULPT design but our major difference stems from our motivation to be able to manipulate the permissions table in user applications with a single write. MMP requires invoking a inter-protection domain call. On a fault, MMP will take the traditional kernel fault, while we strive to fault to a user-defined handler if the access violated user-defined permissions.

Chapter 3

User-Level Page Fault Design

For the remainder of this report, variables proceeded by %, like %AX, will represent the hardware register while AX will represent stored in the register.

3.1 Overview

Interrupts, by our terminology, are events generated by the either the hardware or the operating system upon some event. Handler functions are installed into the interrupt descriptor table (IDT) when the operating system boots. When an interrupt is triggered, the CPU temporarily saves the values of major context registers, (%SS, %ESP, %EFLAGS, %CS, and %EIP) and loads the kernel stack segment (SS0) and stack pointer (ESP0) values into %SS and %ESP, respectively. These values are stored within the Task State Segment (TSS) whose address is stored in the task register (%TR), both of which are written when the kernel boots.

Once the hardware registers hold SS0 and ESP0, all operations will be done in kernel space until the interrupt jumps back to user space. Next, the CPU will push the previously stored user values onto the kernel stack. Depending on the interrupt's cause, it may push an additional error code. Then, the CPU will load the interrupt's CS register and the kernel entry point address, both stored in the interrupt descriptor table when the kernel booted. Until this point, all operations have been performed by the hardware. The kernel entry point for interrupts will push the execution context, i.e. all register values, onto the kernel stack and jump to the specific handler address for the interrupt.

The current system does not lend itself to have low-overhead user-level, hardware-enforced memory protection, for the user cannot access its process page tables directly. Rather, the application must request the operating system to modify paging structures through system calls, such as `mprotect` and `mmap`, with specific permissions.

```
1 void signal_handler(int sig, siginfo_t* info) {
2  void* fault = info->si_addr;
3  mprotect(mm_aligned, 4096, PROT_WRITE);
4 }
5
6
7 void setHandler(void (*handler)(int,siginfo_t *))
8 {
9     struct sigaction action;
```

```
    action.sa_flags = SA_SIGINFO;
    action.sa_sigaction = handler;

    if (sigaction(SIGSEGV, &action, NULL) == -1) {
        perror("sig action err\n");
        _exit(1);
    }
}
int main(int argc, char**argv) {

    setHandler(signal_handler);
    char* mm_addr = malloc((PAGE_SIZE*2 - 1) * sizeof(char));
    mm_aligned = (char *)(((int) mm_addr + PAGE_SIZE-1) & ~(PAGE_SIZE-1));
    mprotect(mm_aligned, PAGE_SIZE, PROT_READ);

  // Regular program code
}
```

Listing 3.1: Signal Handling

To realize the same behavior with a user space handler with read-only memory would require the application to set the initially allocated memory to be read-only through some system call, have the permissions set to read-write within the interrupt handler.

Our goal is to enhance the current memory system with a weaker user-defined permission mechanism. By "weaker," we mean that users can set only stricter permissions than those in kernel page tables. Pages set to read-write in kernel page tables can be set to user read-only in our user page tables. The inverse is not acceptable. If the user program violates user-defined permissions but adheres to kernel-defined permissions, the hardware will raise an exception similar to if the program violated kernel page permissions. However, instead of executing a handler registered in the IDT, the CPU will execute a function address that the user program registered beforehand. After the fault has been handled, the program will continue executing from the point of the fault. Throughout the entire process of raising and handling the fault, no operating system code will be run.

There are also details that we must address. Specifically, after the fault is raised and handled, the CPU will once again attempt to execute the faulting instruction. There must be some mechanism to indicate whether this instruction has executed before so that the execution can proceed. Otherwise the CPU will repeatedly fault and handle the same instruction forever.

3.2 Hardware Modifications

3.2.1 Current Systems

There are several hardware registers that the MMU and the operating system use to coordinate the paging system. %CR3 holds the address of the base of the paging structure (PGD) of the currently running process. In the case of page faults specifically, the faulting instruction will be loaded into %CR2 by the hardware.

Kernel page tables are a hierarchical mapping structure between user-aware virtual memory addresses and hardware-aware physical memory addresses. As it is used to translate every address given by user programs, the (MMU) has specialized logic to minimize translation time. We duplicated most of the same translation logic for traversing the ULI page tables; as such,

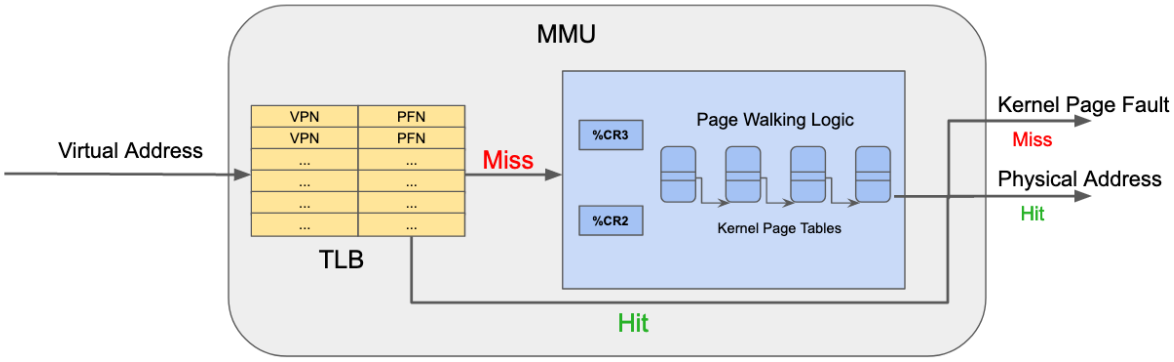

Figure 3.1: General memory translation logic flow

most of the page table structure remained unchanged from current kernel page tables. The only difference in the last level of tables, in which our design differs from current designs. This will be explained in more detail in section 3.4.

The translation look-aside table (TLB) is a cache to reduce the translation latency for the MMU. Standard address translation by page walking requires several memory operations to complete. In most cases, user programs will use only a handful of pages frequently, all which would walk through the same set of page table entries. The TLB cache stores a set of virtual page number to physical page frame translations that can be accessed quickly. The MMU will first reference the TLB to see if the translation is cached and proceed with page walking if is not. (See Figure 3.1.)

3.2.2 Modifications

In total, we add three registers, a write-once bit and user read-only bit in the TLB, and extend MMU logic to implement ULIs.

Registers

We added three registers to support ULIs that parallel the design in current architectures. %CR16 holds the base address of the user page table, analogous to how %CR3 holds the kernel process page table. %CR17 holds the value of the user-level faulting address, analogous to %CR2 during a kernel page fault. %CR18 holds the address of the user-registered ULI handler. The user page table base address and user handler address are saved within the process control block within the kernel and will be swapped in and out upon context switch.

Our first implementation registered the user handler as a new entry within the IDT to parallel current systems completely. However, this will allow only one application to use this mechanism because the IDT is typically written to once during boot time and set as read-only for the remainder of execution. Even if we took away this requirement, we would have to modify this IDT entry on every context switch, which increases the critical path. By holding the handler address

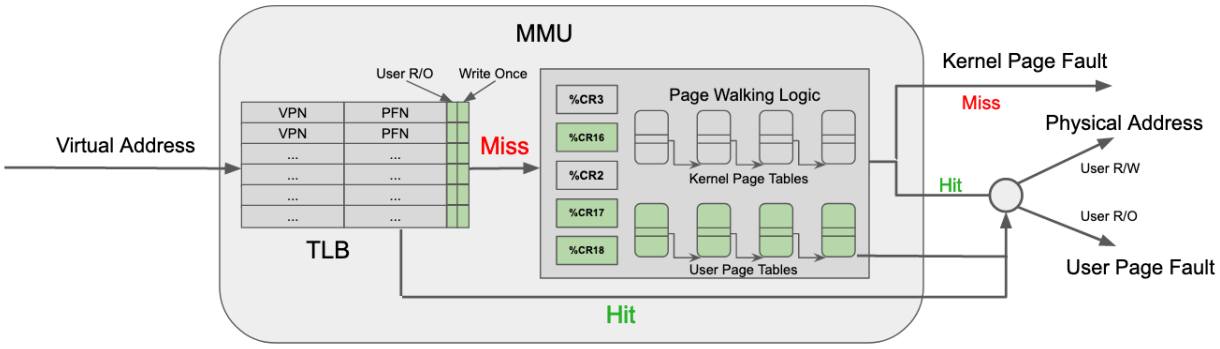

Figure 3.2: Hardware changes for ULPF

in a new register, we increase the number of applications that can use this mechanism at the cost of a register write every context switch. We have also added additional x86 instructions that can read %CR17 and write %CR16, %CR18.

Page Walking Logic

In our design, the MMU will traverse both the kernel and user page tables concurrently 3.3. This way, the addition of this mechanism will not incur an increased critical path length. Unlike walking the kernel page tables, walking the user page tables will never cause a fault for a nonexistent entry. If any mid-level entry in a kernel page table is absent, the MMU will throw either a page fault to bring data into memory or a segmentation fault. In contrast, if an entry is absent in the user-level page table, we silently abort any further translation. If there is an instance where a translation will cause both a kernel and user page fault, we default to the kernel handler.

TLB

We made several modifications to the TLB design to accommodate our mechanism. As our design parallels that of current page tables, we also would like to cache address user-level read-only state within the TLB that would be referenced without having to page walk. Therefore, we added a user-level-read-only bit within the TLB entry. In the event of translating an address not in the TLB, the MMU will walk the page tables and store the page frame number from the kernel page tables and the user level permission from the user page tables in a TLB entry.

However, the MMU will not walk the page tables if the address translation is already cached within the TLB. So, if the address translation for a certain page is already cached with no user permissions and the user set it as read-only in the user page tables, the change will not be reflected in execution as the permission did not change in the TLB. To address this issue, we need a method to allow users to modify a TLB entry corresponding to a virtual page number from user space.

Present methods only provide the instruction `invlpg` to change a TLB entry; it invalidates the TLB entry for a virtual page number and forces a page walk to occur. There are two reasons

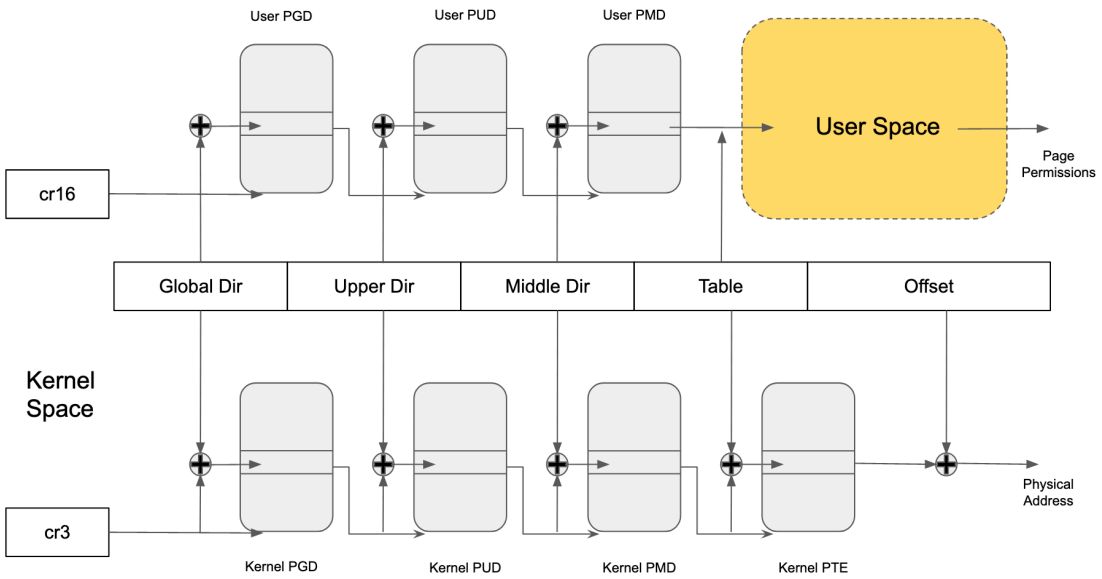

Figure 3.3: Address translation with kernel and user-level page tables

we chose not to utilize this instruction: the first being that this is privileged instruction, the second being that we feel that a user-level operation should not be able to shootdown a system-controlled TLB entry, especially if used in a malicious program that can repeatedly shoot down entries with a single write in user space. While we had the option of creating a user-available `invlpg` to address the first point, we implemented a set of new `x86` instructions, `SET_TLB_RO` and `SET_TLB_RW`, which can be invoked from user space and can only toggle this additional user read-only bit we added in TLB entries. It is the responsibility of the user to invoke these instructions to maintain coherence between the user page tables and the TLB.

Write-Once Mechanism Assuming that the instruction successfully faults into the user space handlre and returns, the hardware will attempt to execute the faulting instruction again. We decided that after a user page fault, the execution should allow the reexecution to proceed, even if the user handler did not explicitly change the page permissions. The current mechanism has no way of determining whether this was the first or second instance of executing this instruction, whether to fault to user space or allow the write. Therefore, we added a second bit, the write-once bit, within the TLB entry that indicates whether the execution should be allowed to continue. The default state for this bit is 0, indicating that this instruction should fault. When the entry is loaded into the TLB from a page walk, this bit is initialized as 0. Upon the first execution instance, this bit is set to 1, indicating that it has executed once, and fault to user space. Upon the second execution, the MMU will see that this bit is 1 in the TLB, and will allow the instruction to execute without faulting. Here, it sets the bit back to 0 so that the next memory operation within the page will also fault.

An issue we saw with this mechanism a single CISC instruction can potentially execute multiple memory operations. For example, the `movups` instructions was two `ldfp`/`stfp` instructions with differing addresses. The first memory microoperation will succeed and the second micro-operation fault. The instruction pointer will revert to the start of the CISC instruction, which will then reexecute the first memory operation, thus leading to infinite looping.

To address this issue, we also save the previously faulting memory address within the TLB logic. Upon executing an instruction, we first check whether the write-once bit is set. If it is not, then the instruction has not faulted before and we need to fault. Here, we also save the faulting memory address. If we see that the write-once bit is set, we know that we have already faulted to on this instruction. However, this instruction may contain microcode that also contain memory references, so we check the faulting address against the previous faulting that we have saved. If the addresses don't match, we continue execution because we have not reached the microinstruction that we faulted on. If the addresses match, we continue execution but reset the write-once bit and clear the saved faulting address so that if the next micro instruction faults, we would be able fault on it. We are basing this logic on the assumption that microoperation sequences will not refer to the same memory instruction more than once. This logic is illustrated in Figure 3.4.

3.3 Software Modifications

Our design on the software front is driven by several goals. First, we are looking for a paging structure that somewhat resembles existing structures to simplify necessary hardware changes. Second, we would like the user to be able to modify the table on a page-granularity without invoke kernel code, meaning that the last level of the tables must be in user space. Finally, we would also like our structures to take up as little space as possible.

Given these requirements, we divided the software modifications into two major portions: the kernel portion and the user library. The kernel portion involves system calls that will install and manage the lower levels of the table in kernel space; the user library will handle the user interface and manage the user space level of the user page table.

3.3.1 Kernel Modifications

Current Design

The kernel is responsible for storing and managing page tables for every spawned process. Page tables are typically copied from the parent process and its contents are replaced with the code and data of the new process. When the process finish executing, its resources, including its page tables, will be cleaned by its parent process and returned to the system. While there are multiple running processes, the kernel scheduler is responsible for context switching between them. From the paging aspect, context switching involves writing the %CR3 register, which holds the base address of the current process' page global directory.

The only way for users to interact with system-level resources, like its page table, is to make system calls, like `mprotect` and `mmap`.

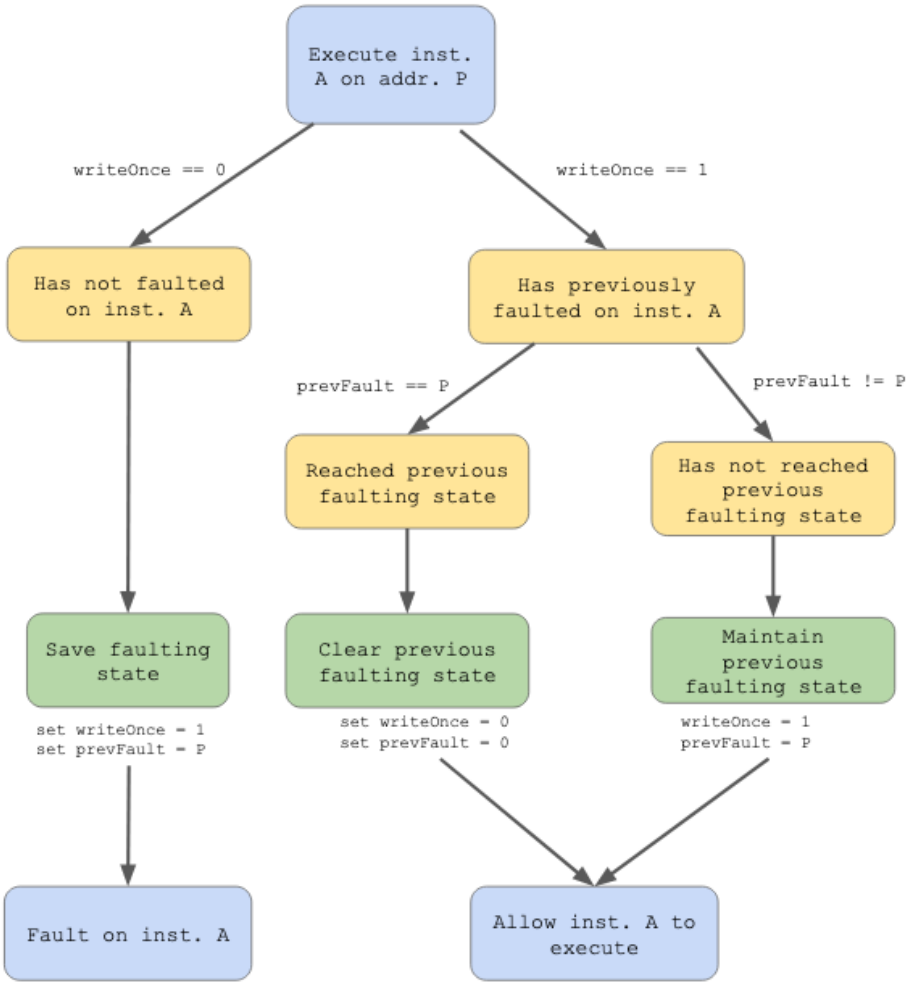

Figure 3.4: Logic diagram for the write-once mechanism. Yellow blocks are commentary; green blocks describes state; blue blocks describes execution.

All kernel modifications were applied to Linux version 2.6.22.9. In total, we added two system calls and extended some existing process structures to support user-level interrupts.

Modifications

There are four levels to kernel page tables to accommodate 64-bit architectures. Since user-level page tables are a per-process structure, we will still like the maximal number of table levels to be in kernel space. So, the first three levels of kernel page tables are duplicated to be the first three levels of user page tables.

The first system call, `long sys_register_uli(void* user_handler_addr)`, makes the hardware aware that the application has activated ULIs. The function will write to CR16 and CR18, respectively holding values for the user-level paging structure and the address of the user-level handler. The process' control block is extended to save the these two addresses. The system call will also allocate the initial level of user level page table, the user PGD, to be in kernel space.

The second system call, `long sys_set_ro(void* vm_addr, void* addr_region)` updates the value corresponding to the virtual address' PMD entry in the kernel page table to map to the given parameter, `addr_region`. The page table indexing and data stored are identical to that of the kernel page tables for the first two levels. In more detail, the given `vm_addr` will be a 48-bit virtual address. In address translation, this address is split into five discreet sections: the first 36 bits are split into four 9-bit sections that are used to index into intermediate page tables. The base address of the following page table level is the indexed entry in the current level. We copy this design exactly for the first two levels: the first 9 bits of the virtual address is used to index into the user PGD to get the base address of the user PUD. The next 9 bits is used to index into the user PMD to get the base address of the user PMD. Here, the designs differ. The user PMD entry will hold the user-provided `addr_region`. This will be explained more in User Library Design. Since this parameter is from user space, it is a virtual address. It must be translated to a physical address to be stored in the user PMD because the hardware assume that table entries are physical addresses.

3.4 User Library Design

The top-most level of the user level page table is in user space, named the page level indicator (PLI). PLIs are a series of PAGE_SIZE-aligned pages that hold the user page permissions for a program using ULIs. Our goal is to have one to two bits in within a PLI to correspond to the user-set permissions for that page.

Continuing where the user page levels left off in kernel space, we are left with the final 9 bits of the PFN to index into this last level. For any first 27 bits of a PFN, or address prefix, will correspond to 512 consecutive pages. If we allocate two bits per page for permissions, we would have 1024 bits, or 128 bytes that would map to an address prefix. A block of these 128 bytes will be referred to as regions. For efficiency, we packed multiple regions, even if they correspond to non-consecutive address prefixes, within a single PLI. Once a new region is allocated, it is passed as an argument to the `sys_set_ro` system call to be stored in the page's user PMD

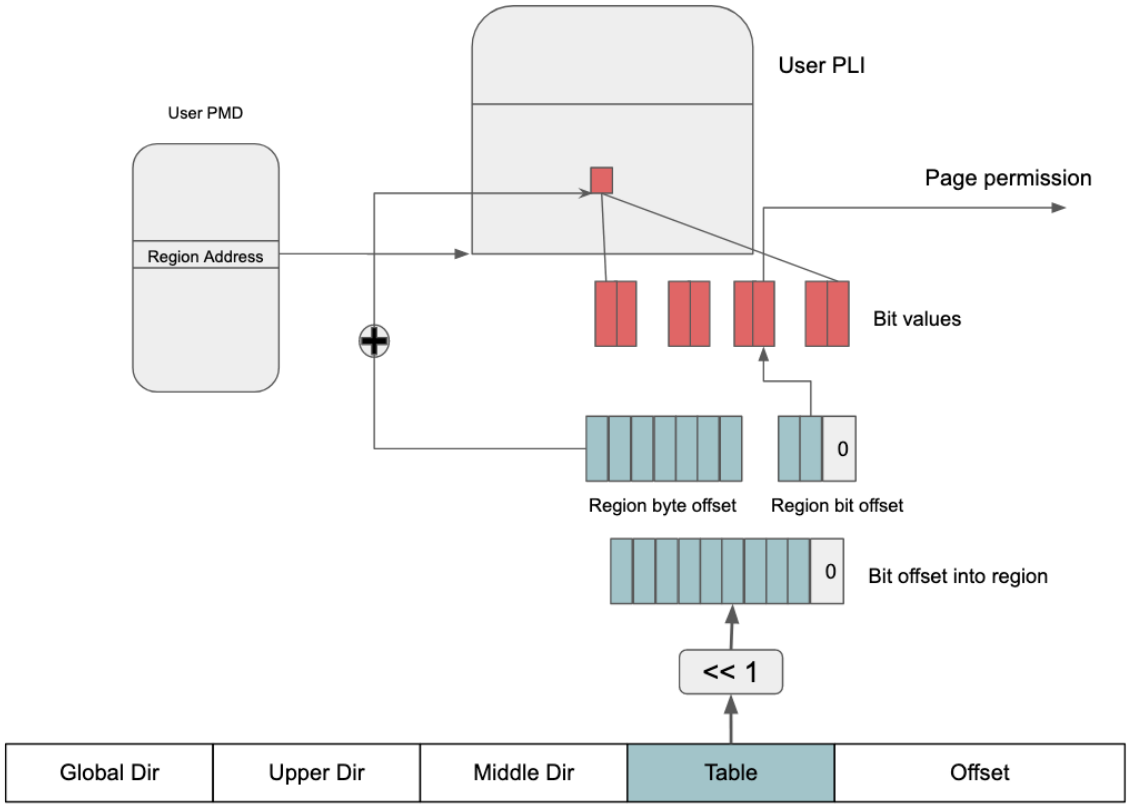

Figure 3.5: Indexing into the PLI using bits 12-20 of the virtual address to get the user-set permissions for the page.

entry. Finally, the last 9 bits of the PFN will index into the region, in 2-bit increments, to set the permissions for the virtual address' page. This is illistrated in Figure 3.5.

The ULPF library will manage the page level indicator (PLI) structures. Internally, the user library has a lookup table that maps an address prefix to its corresponding region address, referred to as the region table. This will reduce the times system call `set_ro` is invoked, only making the call when an address prefix has not been seen before, which means the user_pmd structure must be updated in kernel level anyway.

PLIs are kept in a linked list. The first region of each PLI page has a bitmap indicating which regions within the PLI is available. All allocated PLIs will be checked for free regions before a new one is allocated.

Once the user changes the permission pertaining to any address, the library call must update the TLB to ensure coherence between the user page table structures and the hardware. Here, we invoke the new x86 instructions we mentioned in the TLB section, `SET_TLB_RO` and `SET_TLB_RW`.

3.5 Summary

To realize our design goals for ULPFs, we must modify hardware and software components. We ensure that ULPT permissions will be hardware enforced by basing our design off of existing page walking logic within the MMU. Likewise on the software front, we duplicate the kernel level page table structures for ULPTs to interface with the hardware extensions we made. We move the top-most level of ULPTs to user space so users can modify their memory permissions with a single write instruction within changing privilege levels. We bypass any kernel intervention by modifying the interrupt pipeline to jump to user handler code immediately after the hardware throwing a ULPF.

Chapter 4

Case Study: Watchpoints

4.1 Previous Work

Debugging technologies have always been sought after for its potential for making developers' lives easier, in particular, setting breakpoints and watchpoints. Users can set breakpoints at certain points in there program through a debugger. These are commonly at function or data addresses. When the application runs within the debugger, the debugger will "break" normal execution once it reaches the address the user set and return the control back to the user, often waiting for some user input. For instance, one can set a breakpoint at a function address to see if it ever gets run or break at a certain line and examine the variable state within that scope.

Watchpoints are a special form of breakpoints as it only gets triggered when the value at the breakpoint address changes. Once it is triggered, the debugger will return control to the user. Setting breakpoints and watchpoints are so widely used that hardware manufacturers have included special debug registers to detect whether the breakpoint addresses matches the current memory address. Moving this functionality to hardware greatly reduces the overhead compared to a software implementation.

The most ubiquitous debugging software is GDB, whose variety of functions makes it a popular choice for developers. However, GDB's implementation comes with several drawbacks. While it does use the debugging registers for the first couple of watchpoints, it may either abort or fall back on software watchpoints if the user sets more watchpoints than hardware debug registers. The software watchpoint implementation is very slow, as it forks the application as a child process and uses the `ptrace` system call to single step the application code and check whether the accessed address is watched against some internal data structure. The `ptrace` call also uses signals to communicate between the parent debugger and child application, which adds to the overall slowdown. While this enables setting any number of watchpoints, software watchpoints incur a huge slowdown over the native application.

Researchers are aware of GDB's limitations so there have been many proposals to reduce this overhead. A popular avenue for improvement is using dynamic binary instrumentation (DBI) along with shadow memory to execute a check on each memory access. DBI frameworks are used as program profilers that can observe behavior at runtime. Pin, one DBI framework, takes an executable file as input and generates new code using its internal just-in-time compiler. This

allows Pin to inject user-installed monitoring code before select instructions are executed [3].

Shadow memory is specially allocated memory used to identify permissions for another region of memory. An intuitive example is partitioning the address space in two, with one region being program-accessible and the other being a one-to-one mapping for the permissions of bytes being used. Although space-inefficient, the implementation is straightforward; the instrumentation handler would add a set offset to the accessed memory address to find its corresponding permission in the shadow memory and execute some routine. By this definition, our PLIs are also a from of shadow memory. EDDI [8] uses DBI and shadow memory on this design to efficiently implement unlimited watchpoints in IA32 architecture. They also optimized by having a more efficient memory to shadow memory mapping, register analysis to reduce spilling when executing instrumentation code, and batch checking memory accesses [6].

Along a similar vein, MemTrace [5] expanded on a cross-ISA binary translator to provide memory watching capabilities. It cross compiles between 32-bit and 64-bit architectures and use the additional registers and increased memory space to inject and execute small code sequences within the original executable. The concept is similar to that of EDDI but MemTrace used a binary translator as its vehicle for code insertion. MemTrace also uses shadow memory to cover the original 32-bit application's memory space.

4.2 Modifications

To implement watchpoint functionality with our mechanism, we added a user library that provides an intuitive interface for setting watchpoints.

Inside their application, the user would first register a ULI handler function through the `register_user_handler` call, which invokes the previously mentioned system call, `sys_register_uli()`, to initialize the user level paging in kernel space. The library call also initializes the PLI structures in user space. The user can then set read-only permissions on individual or ranges of addresses that are in its address space through `set_permissions`. For every address that the user sets as read-only through this library call, the library will check whether the address prefix is already in the user level page tables in kernel space by searching through the region table. If it is within the table, then only the PLI structure in user space will need to be updated. Otherwise, the library will invoke system call, `sys_set_ro()` to update the user page tables in kernel space before updating the PLIs and page permissions in user space.

All watched addresses and address ranges are kept in a splay tree, sorted on address. This is necessary because a user-level page fault would occur for every address on a page that contains a watched address. A structure that tracks addresses on a finer level is necessary to distinguish actual watched addresses from false positives. The tree structure dynamically merges overlapping address ranges as the user inserts them to minimize the tree size. This feature has varying results depending on the application. By the nature of the `malloc` library call, the header at the beginning of the allocated memory will prevent merging to occur, even between two consecutive `malloc` calls.

The user handler function will first check whether the faulting address is actually being watched by searching through the watchpoint tree. The faulting address can be accessed by directly invoking the bytecode to read %CR17. If it is a watched address, it will continue execut-

ing the handler. Otherwise, it will immediately return. Either way, the address will be marked as write-once in the TLB, so even if the handler code was empty, execution will progress past the faulting instruction.

```
#define USER_RW     0x0
#define USER_RO     0x1
#define USER_EXE    0x10

int register_user_handler(void* handler_addr);

long set_permissions(void* addr, size_t range, int perms);

unsigned long read_fault_addr();

int lookup_wp();
```

Listing 4.1: Watchpoints API and Definitions

```

#define ulihandler __attribute__ ((user_level_interrupt)) \\
__attribute__ ((disable_tail_calls)) void
ulihandler user_handler() {
    void* addr = read_fault_addr();
    int found = lookup_wp();
    if (found) {
        printf("Watchpoint hit")
    }
    else {
        printf("False positive")
    }
}

int main(int argc, char**argv) {

    int ret = register_user_handler(&user_handler);
    if (ret < 0) {
        return -1;
    }

    char* mm_addr = malloc(PAGE_SIZE * sizeof(char));

    set_permissions(mm_addr, PAGE_SIZE, USER_RO);

    // Regular program code
}
```

Listing 4.2: Example Watchpoint Usage

4.3 Microbenchmarks

The focus of microbenchmarks is to determine the runtime differences between our mechanism and some similar mechanism that current systems support.

4.3.1 Testing Framework

As we explained in the previous chapter, we need to modify the hardware, the operating system, and add a user library. Our hardware modifications were done within the Gem5 simulator. Gem5 is an open-source computer architecture simulator capable of simulating multiple architectures

and different CPU models. We selected this software because it allows us to freely modify internal implementations to fit our needs. It also has a "full-system" emulation setting where it can emulate an unmodified kernel binary. Within its source code, we modified the page walking and TLB execution logic and implemented the additional aforementioned x86 instructions.

To invoke our specificalized instructions, we also modified the LLVM compiler to compile ULI programs. The compiler detects the user handler function through a unique attribute and will automatically generate assembly to save and restore the execution registers prior to, and after the handler code has executed.

Although Gem5 is capable of simulating up to the Linux kernel 3.4, we modified the Linux kernel 2.6 because there are extensive materials that explain its components in depth. Unfortunately, we later found that Gem5 was unable to run some executables because the kernel version was too old. Within the kernel, we duplicated the existing paging system and added the previously mentioned system calls.

For our benchmarks, we selected and wrote programs in C and compiled with Clang 6.0, with our modified LLVM back-end. We compiled everything with `-O3`, unless stated otherwise. We also found that some functions were not supported by the internal `libc` library within Gem5 so all our programs were statically linked against `glib 2.17`. Our cycle counts were taken using the `rdtsc` instruction with the start point after initial variable declaration and argument checking and the end point before any resource cleanup. Our reported cycle counts are the difference between the end point and the start point.

Our user handler is consistent through all benchmarks. It first invokes our library call to check whether the faulting address is a watched address. If it is, the it will increment a global `watchpoint_hit` variable by one, otherwise it will increment a global `false_positive` variable by one.

4.3.2 Programs and Results

Our first microbenchmark is a general overview of the scalability and overhead of the user level interrupt. The program allocates 256 consecutive pages, each 4096 bytes, and sets watchpoints on select addresses according to different distributions. Our measurement covers a loop that writes to each address in order. We also took measurements with two different user handlers - an empty handler and one that searches the watchlist tree to determine whether the faulting address is watched.

There are two different components that influences the runtime. The first is that watching an address will cause the entire page to be read-only. This implies that even watching one address will lead to 4096 faults, of which 4095 are false positives. However this pays off if there are multiple addresses been watched on the same page. In our first distribution, we place watchpoints by randomly selecting addresses and only watching a single byte for each watchpoint, refering to this as the sparse configuration. From the data in Table 4.1, we see that the overall slowdown of the application is linear with respect to number of pages spanned when there is no lookup in the user handler and linear with respect to the number of watchpoints (also the number of nodes within the splay tree) when there is lookup up in the handler 4.1. On average, the slowdown per page watched is 0.7% while the slowdown per node in the watchlist tree is 5%.

Our second watchpoint distrubution involves watching addresses, also in one byte units, each placed with 256 byte offsets of each other. We refer to this the dense configuration. The data in Table 4.2 reflects that the slowdown is independent of the number of addresses watched within the same number of pages spanned. This is exemplified by the discrepancy between the data in Table 4.1 and Table 4.2 in their lookup slowdown. Since the addresses watched in the dense configuration is not consecutive, the number of nodes in the search tree is also equal to the number of watchpoints. However, the slowdown in Table 4.1 for 8 watchpoints is much higher than for 8 watchpoints in Table 4.2 because the tree is searched for 8 pages of memory accesses in the former while only for 1 page in the latter.

Finally, to cover the extremes, we tested a configuration where we watched one address one each page, and one where we watched the entire block of allocated memory. Both configurations will have the same number of pages spanned, leading to the same number of faults. Here, we a slight variation in slowdown for just the mechanism when there are many faults, as shown in the Slowdown (no lookup) column of Table 4.3. However, we also see that the memory accesses from searching the watchlist accumulates even if there is only one node, as seen in the dense configuration. The slowdown for the sparse configuration may be optimistic for we watched the first address on each page and inserted them into the splay tree in the same order as our access pattern. This will lead to fewer number of rotations when balancing.

Table 4.1: Sparsely Distributed Watchpoints

WPs	**Pages Spanned**	**False Positives**	**Slowdown (lookup)**	**Slowdown (no lookup)**
1	1	4095	1.05	1.01
2	2	8190	1.11	1.02
3	3	12285	1.17	1.02
4	4	16380	1.22	1.03
8	8	32760	1.45	1.06
16	16	65520	1.91	1.13
32	32	131040	2.84	1.26
64	55	225216	4.16	1.45
128	101	413568	6.81	1.83
256	156	638720	9.97	2.29

Table 4.2: Densely Distributed Watchpoints

WPs	**Pages Spanned**	**Slowdown (lookup)**	**Slowdown (no lookup)**
8	1	1.05	1.01
16	1	1.05	1.01
24	2	1.11	1.01
32	2	1.11	1.01
40	3	1.16	1.02
48	3	1.16	1.02

Table 4.3: Extreme Distribution of Watchpoints

WPs	Pages Spanned	False Positives	Slowdown (lookup)	Slowdown (no lookup)
256	256	1048320	15.77857026	3.161276259
1048576	256	0	14.47195665	3.253215488

We also replicate the same benchmark using GDB hardware and software watchpoints. In order to have a reasonable runtime, we reduce the overall allocated memory to 8 consecutive pages. As consistent with the sparse configuration, we randomly select a number of addresses to watch in both GDB and user-level page faults. We then time the number of cycles it takes to write to all allocated bytes. For implementation details in GDB, please refer to Appendix A.

Both benchmarks were compiled with `-O0` optimization so that GDB would able to set watchpoints properly without variables being optimized out. This explains why the ULPT slowdown is drastically larger than the slowdowns presented in earlier experiments. GDB only allows setting 3 hardware watchpoints at a time; the overall slowdown to using hardware watchpoints, i.e. the debug registers, is close to negligible. However, if users want to watch more than 3 discrete addresses, GDB software watchpoints will incur an impressive slowdown.

Table 4.4: GDB Comparison

Addresses Watched	GDB HW Slowdown	GDB SW Slowdown	ULPF Slowdown
1	1.05	799600	2.50
2	1.05	1018000	4.21
3	1.03	1203000	6.01
4	–	1446000	6.12
8	–	2101000	8.18
16	–	3773000	12.01
32	–	6546000	17.14
64	–	13520000	19.39

4.4 SPEC Benchmarks

We are trying to use our mechanism on the SPEC2006 benchmark suite. For each benchmark, we chose to only watch all allocated heap memory. Due to several setbacks, we only successfully tested our mechanism on two benchmarks. Some reasons that other benchmarks failed are that the disk image did not have enough memory or the kernel is too old. There are also a couple that did not use heap memory; they mostly accessed global memory, which did not satisfy our experiment requirements.

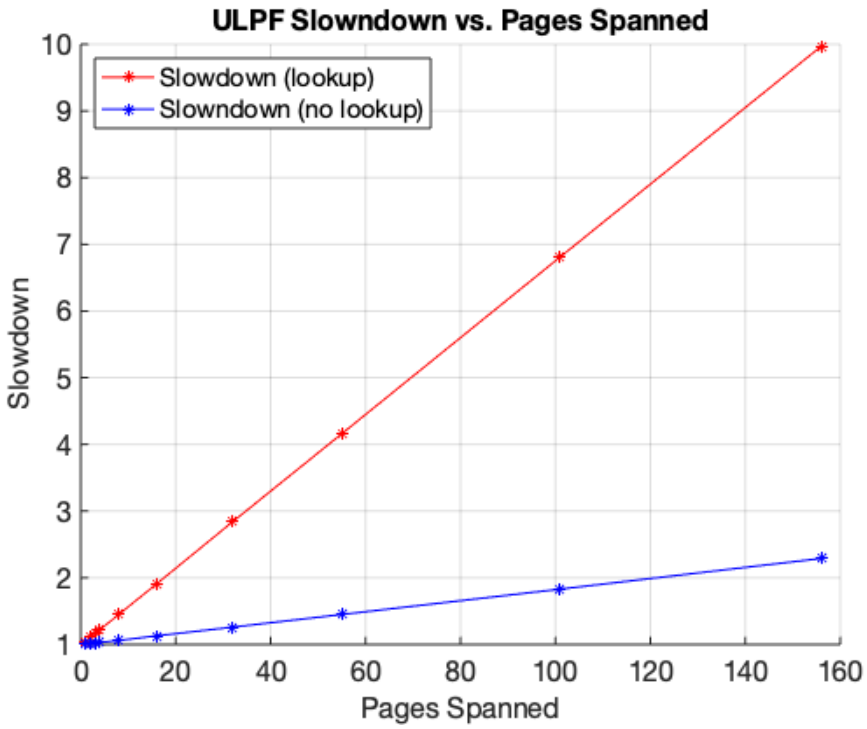

Figure 4.1: Slowdown of GDB software watchpoints vs. pages spanned by watchpoints

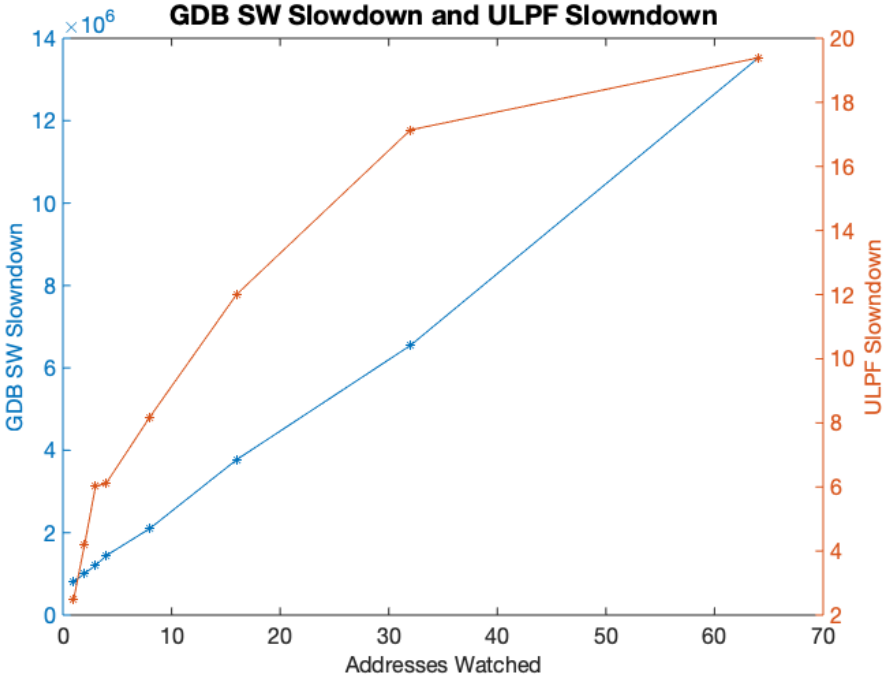

Figure 4.2: Slowdown of GDB software watchpoints vs ULPF

Table 4.5: SPEC Watchpoints

Name	WP Hit	FP	Tree Nodes	Slowdown (lookup)	Slowdown (no lookup)
libquantum	7249859	348	95	6.577591299	1.775092697
lbm	4611909	3930	2	2.122093753	1.070661192

4.5 Summary

From the microbenchmarks, we see both affirmation and issues of our design decisions. The overhead of for the ULPF faulting mechanism is low; there is around a 1% increase in overhead for every discrete page watched. As we show in sparse and dense distribution of watched addresses, this overhead stays consistent. However, when we check whether the faulting address is actually watched using our splay tree, we see that we have a high overhead. This overhead comes from two factors. The first is the number of times the tree is searched; this is proportional to the number of pages spanned. The second is the overall addresses watched; this is the size of our splay tree. The average increase in overhead per watchpoint is 5%. Overall, these results show that our hardware and software mechanisms for ULPFs are providing the low overheads that we hoped. Our design for the watchlist structure needs to improve to reduce overhead of searching.

Chapter 5

Possible Improvements

There are several channels for improving the performance of this mechanism that we did not have a chance to explore.

Another channel may be to do more extensive trials to see which implementation of a watchlist has the lowest average overhead. Our hypothesis of a splay tree having both potential advantages in terms of locality and being balanced were not supported across the board, as a general binary search tree showed less slowdown in some applications.

5.1 Bloom Filters

Instead of a tree structure that holds all watched addresses, there can be an additional level of indirection within the user library that uses bloom filters. Bloom filters are probabilistic data structures that can determine non-membership with certainty. In the case of potential membership, a more concrete data structure, like a tree or hash-map, is needed. However, this method does not lend itself intuitively to represent ranges.

5.2 ULPT Alternative Design

While the decision to have the user page tables span both kernel and user space gives for freedom to the user: updating permissions would not require system calls but only store instructions. However, this simplicity comes with the necessity of all the tables and management structures within our user library, which makes our internal structures vulnerable to stray pointer writes or freeing in the user application. If this occurs, we are saying that it is the user's fault for not using our library properly.

If users are using the given library to implement applications, then the library ensures that the TLB entry will be set to user-read-only upon updating a page's permission. However, if users chose to implement their own library with our system calls, then the user would be responsible for maintaining this coherence. Ideally, we would like to relieve the user of this responsibility, as its absence will cause unexpected behavior and hours of user debugging for it to work.

The crux of the issue lie in our decision to split the ULPTs between user space and kernel space. An alternative design that involves the ULPTs to be entirely in kernel space will resolve

this issue. Users will register a handler function and initialize the user memory tables with the currently implemented system calls. However, updating the page permissions will be handled by the hardware. Users will make a permission update through a special instruction that sets the user-read-only bit for that entry in the TLB. If the page entry is not within the TLB, this instruction performs an `NO-OP`.

This design comes with its own set of coherence issues, particularly when the TLB entry gets evicted. If the entry is mapped to the currently running process, then some mechanism will need to walk the page table to update the user page table so the next page walk will bring in the expected permission. However, if the entry does not belong to the currently running process, then some extra data, like the PLI region physical address, must be stored within the TLB entry to updated before eviction.

This is not a perfect design, just a thought experiment that addresses a detail in the current design.

Chapter 6

Future Work

This iteration of ULPF design is only setting a foundation for future work; there are many issues that are not addressed. We have not discussed whether children processes will inherit their parents' user level handler or whether they will have the same permissions in their corresponding memory also set to read-only. We also have not reflected on whether giving users this functionality will affect program security.

Out of the conditions that we were running applications, we have found that some of our initial design issues no longer applied to, or was not beneficial, to the overall overhead of using watchpoints.

The first was our decision to merge consecutive and overlapping address blocks to decrease the number of node in our watchlist tree. This proved to be useless under our experiment conditions for we chose to watch all allocated heap memory. Given that `malloc` will take a some amount of memory as the header for a block, no consecutive allocated blocks will be merged. There is an argument to be made whether we should account for the header as part of the watched memory and thereby allowing merging watched memory from two consecutive `malloc` calls. However, that this time, there is no method to determine whether a watched address is the return address of a `malloc` call, thus we cannot make this assumption. To accurately implement such a function would require either changes to the library functions or have some structure to track `malloc` return values.

Future work may include to exploring other applications of a user-level faulting mechanism, like stack guarding. We have intentionally allotted two bits for every page in the user page tables to be easily extendable in the future.

Another interesting avenue to explore is having this user level faulting mechanism apply to only certain threads in multithreaded applications. This way, the user would detect whether some thread is writing into another thread's memory or changing some variable value. Unfortunately, that is not easily extendable from our current design for most of the existing hardware only supports differentiating between processes. Since our mechanism builds on existing hardware, we have no way to differentiate between threads of the same process. For example, the MMU will only refer to %CR3, which holds the process page table, for address translation. Only the kernel has thread-level information, but requiring kernel interference will defeat the purpose of ULIs, which explicitly bypasses the kernel.

Chapter 7

Conclusion

We have successfully shown through this design that it is possible to give users the power to define memory protection within their applications. Our approach differs from many previous proposals in that we expand on available architecture designs with hardware logic, rather than limiting our design to software components. We also have a more conservative memory allocation philosophy compared to other shadow memory implementations for we hold the user library to be responsible for detecting false positives. That being said, one major shortcoming of this design is a sub-optimal design for our watchlist implementation. While our initial assumptions for using a splay tree were reasonable at the time, tests on actual applications show that the repetitive splaying for every memory allocation snowballed into a significant overhead. Cutting this overhead down will be a focus for future work.

Chapter 8

Appendix

May it be a light to you in dark places,
when all other lights go out.

Galadriel
The Fellowship of the Ring, J.R.R. Tolkien

8.1 GDB Scripting

For the purpose of benchmarking, we needed such a script to automatically set watchpoints and run the program in its entirety to mimic the behavior of the corresponding ULPT program. GDB can be configured to run a script automatically upon startup. The contents of this script is typically found in `~/.gdbinit`. Each line is a command the user would have input into the GDB interface. Below is an example of a script we used.

```
set pagination off            # Do not limit output length on screen
set logging off               # Turn off logging (can also be set to redirect to another file)
set can-use-hw-watchpoints 0  # Force GDB to use software watchpoints
file gdb_test                 # Name of file to run
break gdb_test.c:24           # Set breakpoint preemptively on line 24 of code file
run                           # Run the program to get variables in scope
watch *(block + 1234)         # block is base address of allocated memory. This sets a
                              # watchpoint on address (block + 1234)
continue                      # Continue with execution
```

Listing 8.1: Sample GDB Script

8.2 Making a New System Call

This is a quick overview of the files needed to be changed to install a new system call, steps to make and install the modified OS, and invoking your new system call in a user application. All this information and more can be found on Google, I hope to make someone's life a bit easier.

My example will be using the Linux 4.19 kernel, more recent kernels will have a similar file structure but the file or variable names might be a bit different. A grep over a classic system

call like `fork` or `wait` would list a superset of the files you would need to change. I will be using Ubuntu 18.04 my distribution. Some Linux distros offer specialized packages to install a modified OS (like `dpkg` for Debian-based distros); it worked once and never again. The method I include below I found to be reliable and should be distro-insensitive but I have only tried on Ubuntu.

Download the kernel source code from the official website and untar it.

```
// Comment: I will be referring to your download directory as $OS_HOME

>> wget https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.19.191.tar.xz
>> tar -xzvf linux-4.19.191.tar.xz

// I will be referring to the linux base directory ($OS_HOME/linux-4.19.191) as $LINUX_HOME
```

The highest level file directory is split into key components of the kernel. I mostly dabbled in `mm` (memory management), `kernel`(most system calls), and a bit in `arch/x86/include/asm` (macros that bridge between the software and the hardware); I don't know much about the rest. While technically we can write system calls anywhere, I'll demonstrate this within the `$LINUX_HOME/kernel` subdirectory because that's where all the cool system calls are.

Within the `$LINUX_HOME/kernel` subdirectory:

```
// my_syscall.c : My first system call
// Super interesting and requires kernel privilege to execute
// Full path: $LINUX_HOME/kernel/my_syscall.c

asmlinkage long sys_mult_five(int input) {
  return 5 * input;
}
```

```
// Makefile
// Full path: $LINUX_HOME/kernel/Makefile

// This is super forgettable and frustrating to debug the resulting make errors.
obj-y = shed.o fork.o exec-domain.o ... \
      ... \
      my_syscall.o
```

Hopping over to `$LINUX_HOME/include`, we need to add your new function to the header files.

```
// syscalls.h
// Full path: $LINUX_HOME/include/linux/syscalls.h

// NOTE: If your syscall takes no parameters, you need to explicitly write void
asmlinkage long fork(void);
...
...
asmlinkage long sys_mult_five(int input);
```

```
// syscalls.h
// Full path: $LINUX_HOME/include/uapi/asm-generic/unistd.h

...
...
// Each syscall is registered with a trap gate, these gates are identified by IDs
#define __NR_statx 291
// This macro will generate and write the trap gate into the IDT for the corresponding ID and
    function
__SYSCALL(__NR_statx, sys_statx)
#define __NR_mult_five 292
__SYSCALL(__NR_mult_five, sys_mult_five);
```

This file defines all system calls and their respective call numbers if invoked from user space.

```
// syscall_64.tbl
// Full path: $LINUX_HOME/arch/x86/entry/syscalls/syscall_64.tbl

...
...
332 common statx sys_statx
// This number is important
333 common mult_five sys_mult_five
```

This is basically it. Time to build.

```
// In $LINUX_HOME, within the shell

// Compile everything
>> make -j $(nproc)

// Install kernel modules
>> sudo make modules_install

// Install the build images into your /boot directory
>> sudo make install

// For Debian-based distros, you don't have to update the GRUB config but if you want:
>> sudo update-initramfs -c -k 4.19.191
>> sudo update-grub
```

I personally found it useful for the boot process to stop at the GRUB menu instead of automatically booting into the first OS.

```
// Full path: /etc/default/grub
...
GRUB_TIMEOUT=-1
...
```

After this change, you must update grub

```
>>  sudo update-grub
```

Great. Reboot and select your new kernel in the GRUB menu (you may have to look in Advanced options and select your kernel version)

```
>>  reboot
```

Now, let's invoke your new system call.

```
// User test for your new system call.
// Full path: $HOME/test_syscall.c

#include<stdio.h>

int main() {
  // syscall invokes system calls by their registered numbers in the syscall_64.tbl file
  int num = syscall(333, 8);
  printf("%d\n", num);
  return 0;
}

// Compile and run
>> ./test_syscall
40
```

Some things to know: These notes are not necessary if your system does not involve any other process or memory components.

- The internal kernel memory allocator is `kmalloc`. The first argument is the same as `malloc`, size of the memory you want to allocate. The second is flags for selecting a pool and/or specific action that you want to execute. For most purposes, this would be `GFP_KERNEL` if you want to allocate pages in kernel space (GFP stands for general free pages). The full documentation is online on the official kernel website.
- `task_struct` is the name of the general process control block. This holds all information (PID, process state, parent process, signal handlers, etc.) pertaining to a process. You can access the current running process by

```
// task_struct is defined below
#include <linux/sched.h>
// get_current() is defined below
#include <asm/current.h>
void fun {
  struct task_struct* proc = get_current();
}
```

- Printing in kernel space uses `printk` with the same syntax as `printf` from `stdio.h`. You can access these logs using

```
>> dmesg
```

  in the shell.

8.3 Good References

- Linux source code explorer: Elixir Bootlin

  Most global variables, macros, and function declarations will hyperlink to a list containing their definition and usage files. You can find the same information through `grep`ping but this website clearly distinguishes the definition files.

  Also contains LLVM, qemu, glibc and several more repositories.
- Building concepts: Operating System Concepts by Abraham Silberschatz, Peter B. Galvin, Greg Gagne

  The classic dinosaur book – not exactly the most practical (look at entry below) but great to get to know the key kernel components.
- Kernel text reference: Understanding the Linux Kernel by Daniel P. Bovet, Marco Cesati

  This book covers, in extreme detail, Linux 2.6. Slightly outdated, but I found this book the perfect bridge between 15-410 and working on an actual kernel.
- Memory Management: Understanding the Virtual Memory Manager by Mel Gorman

  This text provides line-by-line explanations of the memory management system, including page faults, in Linux 2.6.
- The Intel 64 and IA-32 Architectures Software Developer Manuals

  Yes, they are massive. But if you want to play in their sandbox, these explain the rules (to a certain extent) really well.

Easter Eggs

- The Linux kernel usually go be numbers for its release versions, but each release has a name. You can find them in the Makefile of the base directory. A personal favorite: 4.2.8 is "Hurr durr I'ma sheep."
- Developers (Linus) sometimes colorfully gripe about their struggles in the comments. A quick `grep` will find you some interesting bits.

  There seemed to be some dispute between Linus and Sun Microsystems. Prof. Eckhardt might know something about it.
- Due to a change in the Linux Code of Conduct, there has been a patch that replaced all the f-words with "hug." Other profanities have not been adjusted.

Bibliography

[1] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 385–395, 2010. 2.2

[2] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. Providing safe, user space access to fast, solid state disks. *SIGPLAN Not.*, 47(4):387400, March 2012. 2.2

[3] Chi keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *IN PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION*, pages 190–200. ACM Press, 2005. 4.1

[4] Mike Parker. A case for user-level interrupts. *SIGARCH Comput. Archit. News*, 30(3):1718, June 2002. 2.2

[5] Mathias Payer, Enrico Kravina, and Thomas R. Gross. Lightweight memory tracing. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 115–126, San Jose, CA, June 2013. USENIX Association. 4.1

[6] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, page 135148, USA, 2006. IEEE Computer Society. 4.1

[7] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, page 304316, New York, NY, USA, 2002. Association for Computing Machinery. 2.2

[8] Qin Zhao, Rodric Rabbah, Saman Amarasinghe, Larry Rudolph, and Weng-Fai Wong. How to do a million watchpoints: Efficient debugging using dynamic instrumentation. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction*, CC08/ETAPS08, page 147162, Berlin, Heidelberg, 2008. Springer-Verlag. 4.1