OS for Cores without Protection Domains

Mandar Naik, Dr. Mike Ferdman
Computer Science, Stony Brook University
{mnaik, mferdman}@cs.stonybrook.edu

Abstract **- Modern operating systems treat all cores equivalently assuming that each core in the system is a full-fledged core with all protection domains (primarily ring zero and ring three in x86 architecture). This project is aimed at adding an initial effort to augment Linux kernel with support for cores that do not have all protection domains. Building an OS that can handle cores without ring zero will allow experimenting with building heterogeneous systems with simpler cores that don't support ring zero. We have modified Linux kernel to build system call framework, demand paging framework and signal handling framework for such systems. We have designed and tested this kernel on QEMU two cores Linux guest machine where second core was converted to simple core by modifying interrupt subsystem of the Linux kernel. We have evaluated this OS for range of applications from simple to signal intensive application like Tetris [4] although rigorous testing of this OS is still pending. From this project we can prove that it's possible to run Linux kernel on such heterogeneous systems.**

Keywords **– OS- Operating System, Linux Kernel, IPI-Inter Processor Interrupts, APIC – Advanced Programmable Interrupt Controller.**

I. Introduction & Motivation

Many modern CPU architectures include some form of protection domains to protect data and functionality from faults and malicious behavior. This helps in providing different levels of access to separate resources in the computer system. Most popular CPU architecture family, x86, also provides four rings of protection domains from ring zero to ring three. Out of these protection domains ring zero is generally used for kernel mode execution while ring three is used for user mode execution. This project aimed at building an OS for systems which consists of a combination of simple and complex cores. Complex core in the system will have all the protection domains while simple core will only have one protection domain. This implies that only complex core can execute in kernel mode. Most modern OS including Linux kernel assumes that each core in the system has all the protection domains, so such OS can't run directly on these systems without any modification. To the best of our knowledge there is no existing OS that supports such a system. Operating System support for Overlapping ISA [5] paper only discusses scheduling disciplines but did not discuss design and implementation of all subsystem inside an OS. This project is a proof of concept aimed at adding Linux kernel support for such systems.

We believe that such heterogeneous systems will be useful where workload consists of tasks that majorly execute in user space, which could be then offloaded to simple cores. Best example of this could be a web server. Web server executes most of the time in user space e.g. rendering a page, which does not require kernel intervention and only small portion of its execution time is spent in system calls [5]. Main motivation behind building such heterogeneous system would be that one can fit more cores in the system at the same power budget which can lead to increased performance in such systems. Consider a hypothetical system where we can squeeze two cores, one complex and one simple core instead of one complex core. Even if we extract only 75% performance from each of the cores in the new system, we still have 50% overall performance improvement by adding one more core. In order to allow experimenting with such systems it is essential to have an OS that can run on it which will help in measuring performance of such systems. As of now, this OS is targeted exactly for the same use.

This project did not aim at measuring or proving increased performance on such systems but it was aimed at running Linux kernel on such systems. In short, simplicity of the kernel was traded for performance and security aspects of Operating Systems. Main contribution of this project is to get Linux kernel support for the proposed heterogeneous system to allow tasks to run on simple core by –

1) Adding system call execution framework to Linux kernel.
2) Adding demand paging support to Linux kernel.
3) Adding signal handling framework to Linux kernel.

II. Design

Design of any operating system consists of four important pillars – system call execution framework to allow user space tasks to request services from kernel, interrupt subsystem to handle interrupts in the system, demand paging support to handle page faults from user level tasks, and signal handling framework to allow sending and receiving of signals across user processes. This project tries to address these aspects for Linux kernel on proposed heterogeneous systems.

This project was tested on a two core homogeneous system, where we converted the second core to a simple core in Linux

kernel. Ideally, simple core should not enter ring zero at all, so interrupts including timer interrupts, all traps, and fault instructions must be disabled for the second core. We have disabled all interrupts and traps (except for reschedule IPI and page fault) for second core during boot time. We have also disabled timer interrupts (local APIC and broadcast timer interrupts) for second core so that scheduler is not invoked on simple core at all. Currently only restricted tasks can run on second core. Tasks that ran on second core were linked and loaded with our modified musl libc which prevented task from executing system call using any of the kernel trap instructions.

Linux kernel boot parameter "isolcpus" was used to isolate second core so that second core was totally isolated from other cores and prevented kernel on first core to do any load balancing. Thus, we converted second core to a simple core on the homogenous system. For further discussion we will refer second core as simple core while first core which can run in ring zero as complex core. Now we discuss design of each of the framework added in the system below –

1) **System Call Framework**:
System call execution forms the most basic need for any user space task. Task that intends to execute on simple core will never be able to execute in ring zero so it will never be able to execute system call using regular CPU trap instructions. This requires us to have a separate support for system call execution for such tasks.

We could think of mainly three approaches for executing system calls from tasks on simple core. First for each system call execution task on simple core can be moved from simple core to complex core, execute system call on complex core and then it can return back to simple core. Second approach was to convey system call arguments to some user level task running on complex core using shared memory or message passing which then would execute system call on behalf of the task on simple core. Third approach was to create a kernel thread from boot time which would be solely responsible for executing system calls on behalf of task on simple core. In this case system call arguments can be conveyed through shared memory and result can be returned through shared memory.

We selected third approach for our design. Second approach is very similar to third approach but third approach is simpler as kernel thread has access to kernel subsystem while a user task does not. First approach has inherent disadvantage of explicit moving of tasks across cores twice for each system call and so we did not pick up this approach.

System call is always executed on behalf of a task so executing new system call using kernel thread should be able to address this concern. We could think of two approaches for task impersonation. First approach was to copy fields of target process descriptor to kernel thread descriptor and thus create an impersonation. Second approach was to temporarily switch the context of a process running on complex core to the context of task on simple core and then perform system call on complex core. We chose first approach to simplify total design. Context switch approach is little tricky to get working as it involves run queue lock that is acquired by one process to be released by new process that gets scheduled on the core.

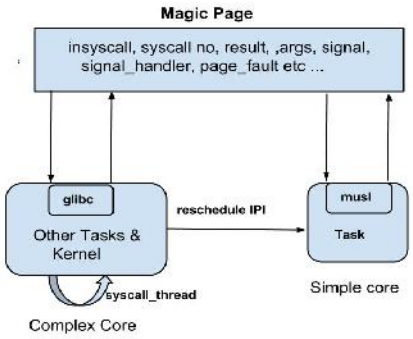

Figure 1: System Call Execution Framework

Figure 1 explains system call execution framework used in our OS. In modified Linux we have mapped a page called magic page in each process at fixed virtual address right from boot time which was used for passing system call arguments from task on simple core. Kernel thread called syscall_thread was created during boot time and suspended until it receives system call execution request from simple core. Scheduler on complex core monitors if there is any pending system call request from task on simple core. If there is such a request it impersonates syscall_thread with task on simple core and wakes it up. After waking up syscall_thread copies system call arguments from magic page and executes system call. After system call execution is complete result of the system call is returned through magic page itself. This represents a typical system call execution on our OS.

Here "syscall_thread" is set with CPU affinity of core zero so that it always executes on complex core. In order to allow task on simple core to issue system call without using any kernel trap instruction task is linked and loaded against a modified musl-libc [2]. We changed musl-libc to invoke system call by copying arguments to magic page and made to spin until syscall_thread completes system call execution on complex core. Also, modified libc takes care of transferring control to signal handler if there is any pending signal in magic page. Signal handling is explained in more detail in subsection 3. Only programs compiled and linked against modified musl-libc were allowed to execute on CPU1.

Exit system call execution is the only exception to this design as if syscall_thread executes exit system call then it can cause syscall_thread itself to exit which is not a desirable effect. As per our design syscall_thread should run for the lifetime of a session once it's created. Killing task on simple core in response to exit system call will need to force another task on simple core. Mechanism of forcing new task on simple core was not as per timeline of this project. Hence to make things

simpler when kernel scheduler on complex core detects exit system calls it is made to send a reschedule IPI to simple core which causes simple core to enter ring zero. Our modified reschedule interrupt handler then suspends task on simple core, changes its cpumask to point to complex core and then next task gets scheduled on simple core by kernel scheduler. As a result task migrates to complex core and it resumes its execution on complex core to again execute exit system call but this time using "syscall" trap instruction. This is the only case where kernel scheduler runs on simple core in this OS. Reschedule IPI backdoor is only for temporary use and will be replaced in future with custom IPI which will be responsible for scheduling new task on simple core from complex core. Further discussion of this idea is out of the scope for current project.

2) **Memory Management Subsystem**:
Demand paging is one of the most important features provided by almost all modern operating systems including Linux. Typically Linux kernel allocates VMAs in response to memory allocation requests from user space tasks and actual page table modifications are deferred until task refers the same memory region for first time by handling page fault i.e. memory gets allocated on demand. This access for unmapped page leaves faulting address in cr2 register as per x86 architecture. This address is read by page fault handler to install page table modifications. It's worth noting that in x86 architecture cr2 register value can only be read from ring zero. As we had simulated simple core on x86 homogeneous system we needed to take care of this constraint while designing page fault handler for simple core.

In our OS for page faults on simple core task enters ring zero, copies address and other related parameters to magic page and returns to user space. Kernel scheduler on complex core detects page fault from simple core using magic page field and wakes up syscall_thread on complex core. "syscall_thread" on complex core then handles page fault while impersonating task on simple core. Until page fault gets handled on complex core simple core keeps on faulting on the same address. Once syscall_thread completes page fault handling, task on simple core resumes its normal execution.

Please do note that this page fault design is a way of getting around x86 architecture constraints that only ring zero can read cr2 register value. While in proposed heterogeneous system simple core can read cr2 register value and this will not be a concern since simple core can't execute in kernel mode. However further discussion on this topic is out of the scope of current project.

3) **Signal Subsystem Framework**:
Signal is an important feature introduced by UNIX OS family which allows processes to communicate with each other and kernel. In Linux kernel, kernel generates and handles signal in ring zero. When task executes any system call on its return path kernel checks if there is any pending signal to be handled and then it changes user stack to change instruction pointer to point to signal handler. This causes task to execute signal handler in user space and after completion of signal handler execution control returns to instruction next to system call in program. Return from signal handler is handled in Linux kernel using sigreturn system call which causes task to return in kernel mode after executing signal handler. Linux Kernel then changes user space stack to get back previous user space context.

In our OS we have simplified signal handling design by avoiding use of sigreturn system call on return path of a signal handler. Signals for the task on simple core are delivered to syscall_thread but syscall_thread does not handle signals through Linux kernel signal handling framework as kernel thread does not and should not jump to user space. Pending signal for task on simple core is handled after task on simple core completes next system call execution. So if syscall_thread finds that there is a pending signal then it removes signal from pending signal queue and writes signal number, signal handler to magic page. As syscall_thread impersonates the task on simple core private as well as shared pending signal queue of syscall_thread is same as that of task on simple core.

When a process sends a nonfatal signal to task on simple core or when kernel generates a signal for task on simple core signal actually is redirected to syscall_thread. So if syscall_thread is executing non-blocking system call then it handles signal after system call execution is complete. If system call is a blocking system call then syscall_thread gets interrupted during system call execution returning error result as expected. Once syscall_thread dequeues signal and copies signal number, signal handler in magic page it sets field in magic page to indicate completion of system call request. Task on simple core which was waiting for system call execution result checks if there is pending signal in magic page, if yes it jumps to signal handler from musl-libc. Thus user space stack is manipulated automatically by user space jump. Fatal signals like SIGKILL are handled immediately by exit system call framework mentioned above.

There is another interesting case of system call execution from signal handler context. Signal subsystem should support system call execution during signal handler execution. This case gets handled through the same signaling/system call framework mentioned above. Inside signal handler we support only signal safe system calls as per POSIX standard [1]. Any other system calls executed from signal handler exhibits undefined behavior.

III. IMPLEMENTATION

Implementation for this project spans mainly in early kernel initialization, scheduler, process and signal subsystem of Linux kernel. Total of 1057 lines of changes were required in Linux kernel and musl libc. Git source version control tool was used for managing this project. We discuss changes for each subsystem in this section.

1) **Boot or Kernel Initialization**:
In this section we discuss changes required for converting second core to a simple core on homogeneous system where each core has all protection domains. First of all, during boot time we passed "isolcpus=1" kernel parameter to isolate second core from other cores in the system. "isolcpus" kernel parameter disables load balancing on simple core and makes sure that no task gets executed on simple core except for the tasks invoked explicitly using "taskset" command. So apart from per CPU idle threads no task was scheduled on simple core. Each task is executed in Linux kernel using cpumask with all bits set in bitmap to indicate that it can be scheduled on any core during its lifetime. But as load balancing is disabled only tasks with specific cpumask for simple core were executed on simple core. "taskset" utility spawns a task by sending reschedule IPI to simple core which then schedules task on simple core. As of now page fault handler and reschedule interrupt are the only backdoors for simple core to enter kernel mode which will be replaced in the future by custom IPI.

During kernel initialization we disabled all interrupts including traps ("trap_init" and "early_trap_init"), timer interrupts ("__setup_APIC_LVTT") for simple core. Disabling timer interrupt made sure that scheduler was not called on simple core per tick thus preventing it from entering ring zero per tick. Disabling local APIC timer interrupt for simple core involved masking of corresponding entry in LVT-Local Vector Table. Most of these changes were implemented in "__start_kernel" function in init/main.c where all resources are initialized in the kernel.

2) **Process Subsystem**:
"syscall_thread" was created during kernel initialization phase when any process in the system does an exec. "syscall_thread" sets its CPU affinity to complex core in the system. Kernel thread does not have its own context but always runs in the context of last running task. Same is the case with the syscall_thread. Also a physical page was allocated during boot time called magic page and mapped in all processes on page immediately after "vsyscall" page. This page gets mapped during exec operation because exec starts the new context of process. "vsyscall" page gets mapped in each executable at fixed virtual address 0xffffffffff600000. So it's rare for any task to map any page at virtual address 0xffffffffff601000 so we chose this location for mapping magic page. Our OS detects any process scheduled for simple core using "cpus_allowed" field in process descriptor (task_struct). Most of these changes were done in "do_execve_common" function in fs/exec.c file.

Syscall_thread executed system call by moving system call argument values from magic page to required registers. After returning from system call "rax" register value was copied to magic page in "res" field. On return path syscall_thread checked if there is any pending signal for task on simple core, if yes it copied signal number and signal handler into magic page. As magic page was mapped in all processes any process could wakeup syscall_theead so scheduler did not need to wait for certain process to get scheduled to check for any pending system requests. After system call execution is completed syscall_thread again suspends itself until next system call request comes.

Typical structure for magic page content is as shown in figure 2 –

```
#define SHMADDR (0xffffffffff601000)
struct __cse502_syscall_struct_t {
    volatile int insyscall; /*System call request */
    long n;/*System call number */
    long arg[6];/*System call arguments */
    unsigned long res;/*System call result */
    volatile int exit;/* needs to call exit */
    volatile int signal;/* signal number */
    void (*cse502_sighandler) (int);/*signal handler*/
    //page fault arguments
    volatile int page_fault;/* page fault request */
    unsigned long fault_address;/* page fault address */
    unsigned long error_code;/* error code */
    unsigned int flags; /* flags register value */
};
```

Figure 2: Magic page structure

Impersonation of task: Impersonation of task on simple core by syscall_thread was very important part of system call execution framework. We achieved this impersonation by simply copying necessary fields from process descriptor of task on simple core to syscall_thread process descriptor e.g. "mm", "active_mm", "fs", "files", "signal" etc. Here we merely copied field pointers but did not increase any reference count for fields inside process descriptors because we did not want syscall_thread to permanently share these fields. Those fields are used only for temporary impersonation.

Before task on simpl core can execute exit system call from complex core, context of syscall_thread is saved back to its original context. This made sure that we can use same syscall_thread for the lifetime of a session.

3) **Signal Subsystem**:
This section describes changes needed for implementing signal handling in this OS. "syscall_thread" impersonates task on simple core and it was made to receive signals for task on simple core. A new function (__cse502_handle_signal) was implemented based on the way signals are checked and dequeued from pending signal queue of a process inside "get_signal_to_deliver" function of Linux kernel.

The "__cse502_handle_signal" function checks pending signal, removes a signal from pending queue if the signal is not ignored by a process. It then copies signal handler and signal number to magic page and if system call execution was interrupted by a signal then it sets system call results to error value where typical error value is -EINTR.

For fatal signals intended for task on simple core like SIGKILL kernel executing on complex core sent a reschedule IPI to simple core. This caused our modified reschedule IPI handler to be called on simple core which follows exit system call setup to terminate the task.

4) **Demand Paging**:
It is ideal for simple core to have minimal kernel intervention on receiving a page fault. We implemented page fault handler in our OS by writing a new function based on page fault handler of Linux kernel. Linux page fault handler was modified such that task on simple core after receiving page fault copied parameters required for processing page fault to the magic page.

Our new modified page fault handler "__cse502_do_page_fault" reused Linux kernel page fault handler as much as possible by skipping any kernel space page fault handling cases and used page fault arguments from magic page.

4) **Musl libc**:
In order to avoid task on simple core to invoke any system call using direct system call trap instructions like "int x80" or "syscall" we decided to modify musl libc which was linked and loaded with task on simple core. We chose musl libc as it is supposed to be lightweight, simple and robust.

Musl-libc modifications mainly included changing system call interface for x86_64 in musl. There are two interfaces with which musl invokes system call, one is architecture specific and other is internal POSIX signal handling specific interface. We have modified both interface (arch/x86_64/syscall_arch.h and src/internal/syscall.h) to point to same modified interface which makes them copy arguments to magic page instead of making system calls using "syscall" instructions. After copying parameters process task is made to spin unless it receives system call result from syscall_thread on complex core. After system call execution is finished task checks whether it needs to call exit system call or it needs to call any signal handler. If yes task does so and finally returns system call result back to the program.

IV. EVALUATION

This OS was tested with the help of QEMU running guest Linux with two cores emulation paired along with the gdb to debug Linux kernel.

This section tries to answer following questions –

1) Is it possible to run simple tasks on simple core using modified system call framework?
2) Is it possible to run signal intensive tasks on simple core?

We tested if second core simulation to simple core is handled exactly as we expect by running system utilities inside Linux guest on QEMU. For instance "cat /proc/interrupts" showed number of timer interrupts handled for second core as zero. "top" showed second core as 100% idle. These results were matching with the expected features of simple core.

Main aim of this project is to prove that it's possible to run tasks on a system where there is a combination of types of cores. Some cores will have all protection domains while few cores will not have any protection domain. So we have tested multiple programs ranging from simple programs such as "hello world" to complex signal intensive programs like Tetris grame [4]. Tetris game was a good test case for signal subsystem support in our system because of signal intensive nature of the program.

Apart from Tetris we also tested programs which test the system call execution from signal handler context. During this testing we found that there is undefined behavior if any unsafe system call is invoked from signal handler. This is because signal handler represents the asynchronous execution which might interrupt the state maintained at libc like buffer positions, heads etc. So we observed that our signaling framework can only support POSIX standard signal safe system call execution from any signal handler context.

For testing fatal signals (SIGKILL, SIGSTOP) in the system we tested termination of programs using Ctrl-c and stopping programs using Ctrl-Z. Ctrl-c terminates task as expected but Ctrl-z stops syscall_thread instead of stopping task on simple core.

Our evaluation suggested that there is significant performance overhead in executing task on simple core mainly because of the way page faults were handled in the system. Page fault handler performance was visibly poor. Actual system calls execute almost as fast as normal Linux system. As performance of the system was not the main goal of the system we do not have any empirical measurements for this system but our evaluation definitely confirmed that it's possible to run Linux kernel on such heterogeneous systems and support execution of tasks on simple cores. For most updated status of on evaluation please refer to COMPAS webpage on heterogeneous cores [3].

V. CHALLENGES

We faced number of challenges to modify Linux kernel to support execution of task on new simpler cores. Here we describe some of the biggest challenges involved in this project.

Converting second core of homogeneous system to simple core was challenging as we needed to disable timer interrupts on second core. This task became more difficult with gdb being not able to transition to long mode during boot time. We could have possibly used other source level debugger like kgdb but kgdb also involves complex setup. So here we somehow managed with printk's to debug Linux kernel during boot time.

Mapping magic page in process address space was important aspect of system call framework. Initially we intended to

extend VSYSCALL page by one page and use second page as magic page. But this was very tricky task as physical page for VSYSCALL is reserved using labels in assembly source code and if one tries to change it then there are lots of adhoc checks throughout kernel which start to fail. Hence we decided to drop this idea and instead fix virtual location to page after vsyscall page (VSYSCALL_START + PAGE_SIZE) and allocate a random physical page each time system boots.

Another important challenge was executing exit system call on simple core. Exit system call caused "syscall_thread" to exit which was not desirable. Hence we modified reschedule IPI to support exit system call.

One more change that we found very challenging was delivering signal to syscall_thread during blocking system call execution to interrupt system call execution. If signals are not targeted to syscall_thread then system call execution can't be interrupted. We solved this problem by changing current target of signals for task on simple core to "syscall_thread" and change pending signal flag (TIF_SIGPENDING) of "syscall_thread" instead of task on simple core.

VI. Future Work

This OS currently supports two cores where only one core can be the simple core. Multi-processor support where multiple simple cores can coexist is still pending. Supporting this would need dividing magic page into multiple sections one for each simple core and spawning one kernel thread per simple core so that system calls can be executed independently on complex cores.

Currently multi-threaded programs do not work with current system call framework. If task on simple core forks new thread then it also tries to execute on same simple core as it inherits cpumask from parent thread. But as parent does not relinquish simple core until its execution is complete and waits at the same time for child to finish its execution it causes system deadlock. One of the solutions that we tried to solve this problem is setting cpumask of new thread to first core. But as this thread shared modified mul-libc with parent we need to schedule this to another simple core in the system. To solve this problem we will first need multi-processor support for our OS or we will need to eliminate musl libc and propose a new framework for system call execution.

To spawn new task on simple core we use "taskset" utility and reschedule IPI in kernel. This is actually a temporary solution to check basic functioning of the system. Support for adding custom IPI to spawn a new task on simple core without calling schedule is yet pending. One can use message passing interface used in Barrelfish paper [6] to implement this in future.

Currently system can boot only when network support is disabled. Exact cause of this is yet unknown but this should be fixed in the future.

Currently signal subsystem handles signal only after next system call is executed so there is no pending list of signals maintained in the magic page. By current design nonfatal signals are not handled immediately for task on simple core. Decision on when this needs to be supported and exact design on how to handle this situation is not yet decided. Also, once we get OS up and running for all above cases we plan to improve performance and security aspects of this OS.

VII. Conclusion

From this project implementation and its evaluation using range of simple to complex programs it has been proved that it's possible to run Linux kernel on system which consists of simpler cores. Changing Linux kernel to support new system call execution framework, demand paging framework and signal subsystem for new simpler cores was a challenging task. Currently page fault handler in this system has significant performance overhead which limits the performance of tasks running on simple cores. But we believe that this overhead can be eliminated in the future by making sure that syscall_thread gets scheduled immediately after there is pending page fault from simple core. At last I can conclude here that it's possible to run Linux kernel on heterogeneous system with a combination simple and complex core.

VI. References

1) Signal(7)- Linux man page POSIX standard - http://linux.die.net/man/7/signal

2) Musl – new standard library for linux based devices - http://www.musl-libc.org/.

3) COMPAS OS For Cores without Protection Domains – http://compas.cs.stonybrook.edu/projects/hive/

4) Micro Tetris, based on an obfuscated tetris, 1989 IOCCC Best Game by John Tromp and Joachim Nilsson- https://github.com/troglobit/tetris.

5) Jeffrey C. Mogul et. al., "Using Asymmetric Single-ISA CMPs to save energy on Operating Systems", Micro IEEE May-June 2008 - http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4550858.

6) Andrew Baumann et. al., "The Multikernel: A new OS Architecture for scalable multi-core systems", SOSP 2009, http://www.barrelfish.org/barrelfish_sosp09.pdf.