gemOS: Bridging the Gap between Architecture and Operating System in Computer System Education

Debadatta Mishra
Computer Science and Engineering
Indian Institute of Technology Kanpur, India
deba@cse.iitk.ac.in

ABSTRACT

Providing adequate exposure to architecture and OS interfaces can enable the students with better understanding of the concepts and increase their interest towards research problems crossing the hardware-software boundaries. Moreover, a framework to explore possible enhancements spanning across the architecture and operating system (OS) layers can be very useful for researchers. The existing tools and techniques used to teach system courses like OS and Computer Architecture serve the objective of respective courses to a large extent. However, exploration crossing the hardware and OS boundaries is particularly difficult due to lack of proper teaching infrastructure integrating the two layers. In this paper, we propose to use a specialized OS executing on the Gem5 full system simulator as an alternate approach to explore the architecture and OS boundaries. Open source OSes like Linux are not suitable for this purpose simply because they are not designed to be used as teaching infrastructure, especially on a system simulator like Gem5. We propose gemOS, a simple OS for 64-bit X86 simulation platform provided by the Gem5 architectural simulator. Further, we propose a simple framework using gemOS along with Gem5 and show its utility towards understanding the OS and architecture interactions, and, show the quick prototyping support of the framework to implement innovative ideas spanning across the two layers. We present use-cases and examples to demonstrate the utility of the proposed framework as a teaching and research infrastructure.

CCS CONCEPTS

• **Applied computing** → **Education**; • **Computer systems organization** → **Architectures**; • **Software and its engineering** → **Operating systems**.

ACM Reference Format:
Debadatta Mishra. 2019. gemOS: Bridging the Gap between Architecture and Operating System in Computer System Education. In *Workshop on Computer Architecture Education (WCAE'19), June 22, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3338698.3338887

WCAE'19, June 22, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6842-1/19/06…$15.00
https://doi.org/10.1145/3338698.3338887

1 INTRODUCTION

Innovations crossing the boundaries of hardware and software layers of a computer system are believed to be one of the focus areas in the coming years [14]. Moreover, specialized computing platforms are becoming very popular with advancement of technologies like Internet of Things (IoT) and System on Chips (SoC). Students and prospective researchers are expected to gain clear understanding of the low-level hardware (e.g., micro-architecture) and software (e.g., OS and hypervisors) abstractions, their interfacing mechanisms and the cross-layer impacts with respect to performance, security etc. One step in this direction is to explore the cross-layer design space as part of the system courses in computer science curriculums. These courses should be designed to enhance holistic understanding of the systems which in turn can open up innovations in the hardware-software co-design space to a large extent.

To explore computer organization and architecture with a practical perspective, a variety of tools and infrastructures are used by educators world-wide. For example, to teach computer organization and its building blocks like sequential and combinatorial logic, FPGA based teaching aids are predominantly used. Similarly, for better understanding of micro-architectural design paradigms like instruction level parallelism (ILP), caches and virtual memory etc., software simulators like Gem5 [12], Champsim [2] etc. are considered to be standards. In the operating system pedagogy arena, many teaching operating systems like xv6 [7], pintos [5], nachos [13] etc. are commonly used. While the above utilities can address the requirements of individual subject areas, they fail to provide adequate support to understand the architecture-OS interfaces and encourage designing solutions crossing the boundaries of the two layers because of the following reasons.

First, there are very few full system simulators like Gem5 [12] which can be used to explore the system design space in a holistic fashion, especially for closed source commercial processors like Intel X86. For example, accurate simulation of the memory management unit (MMU)—the page table walker (PTW), the translation look-aside buffer (TLB) etc., requires comprehensive simulation platform like Gem5. Further, executing an OS with detailed full system simulators like Gem5 [12] is not straightforward and requires special modifications to Gem5. For example, Gem5 full system mode supports only 64-bit simulation for Intel X86 architecture, where most of the teaching OSes are written for 32-bit X86 systems. In the current Gem5 X86 simulation platform, only the Linux kernel can be used. Moreover, simulation of multi-core Linux kernel on Gem5 is non-trivial and requires specialized kernel configurations.

Second, usage of full fledged open source OS like Linux for education and quick prototyping is not trivial. This is primarily because of two reasons—(i) The Linux kernel code base is very vast

Teaching OS for OS and architecture clases	Gem5 support	Boot time	OS complexity	Suitability for cross-layer exploration
Teaching OSes (xv6, pintos etc.)	×	fast (on Qemu)	small code base, well documented	not easy, inflexible
Linux	✓	very slow (~10mins)	large code base, complex features	not easy for beginners
Ideal	✓	fast (~ 1min), multi-staged	small code base targeted for co-design	tailer-made and easy, Gem5 advantages

Table 1: Comparison of alternative approaches to meet the HW/SW co-design infrastructure requirements.

(~16 million LOC) which is not easy to understand and modify, at least at the beginners level, (ii) Testing modified OS features using Linux consumes a lot of time as the booting time of Linux on Gem5 simulator is painfully high. Note that, checkpoint support of Gem5 [12] is not useful when the OS itself is modified along with the underlying architecture during the development of prototype solutions or solving assignments.

Third, while system emulators and virtualization solutions (e.g., Qemu [11] etc.) provide an efficient alternative for OS development [20], the emulators are designed with a completely different objective altogether. For example, Qemu tries its best to execute the code in a native manner as much as possible and avoids the trap-and-emulate model by employing techniques like binary translation [8]. Moreover, modifying the underlying hardware micro-architectures (e.g., virtual memory etc.) in Qemu is non-trivial. For better understanding of architectural designs and its impact on OS efficiency, we need quick and dirty prototyping support along with necessary architectural statistics and deterministic OS level behavior. Emulators like Qemu are not very suitable to meet the above requirements.

To address the above challenges, we propose gemOS, a small 64-bit OS for X86 architecture, *custom-designed for Gem5 architectural simulator*. Being very lightweight, gemOS can boot very fast compared to the Linux kernel and can be modified easily because of its simple design. Moreover, with Gem5, debugging the OS using the underlying architectural information (using the dual debug mode as explained in Section 3) is possible which offers additional flexibility to the students to make use of it for OS courses. All the features of Gem5 architectural simulator—flexible hardware organization capability, comprehensive and extensible architectural counters and, a detailed debug framework—are reusable in the proposed gemOS-on-Gem5 infrastructure. The proposed platform can serve multiple purposes in academic setups: (i) as a supplement to OS courses to design OS assignments for lab exercises, (ii) for advanced systems courses designed with the objective of improving the clarity of architecture-OS interfaces and exploring innovations in the intersecting areas, (iii) as a tools for quick prototyping of new ideas in the early stages and verification of hypotheses in research pursuits.

The summary of our contributions are as follows,

- We propose gemOS[1], a simple lightweight OS for 64-bit X86 platform tailor-made to execute on the Gem5 architectural simulator.
- We present the advantages of using gemOS on Gem5 as an academic infrastructure where the benefits of both the comprehensive simulation capability of Gem5 and the specialized design of gemOS can be leveraged.
- Using illustrations, we showcase the applicability of the proposed platform in achieving the objectives like better understanding of the OS-architecture interfaces, quick prototyping and analysis of new features and the support for implementing solutions crossing the hardware-software boundaries.

The paper is organized as follows. We present the motivation and the requirements to design a light-weight OS for Gem5 simulated 64-bit X86 platform in Section 2. In Section 3, we present the high-level overview of gemOS architectures, explain the important features and the OS implementation details. We demonstrate the usage of gemOS through couple of illustrations in Section 4. The work is concluded along with a discussion regarding the future directions in Section 6.

2 BACKGROUND AND MOTIVATION

2.1 Gem5 architectural simulator

The Gem5 architectural simulator is a discrete event simulator which provides extensive and detailed simulation of several architectures including X86, ARM, ALPHA etc. It is one of the widely used open source architectural simulators with full system simulation capability and can execute a full fledged OS kernel along with device support. This is particularly useful to develop new micro-architectural features and realize the benefits by modifying the kernel. Gem5 supports Linux kernel on simulation platforms like X86, ARM and ALPHA. One of the limitation related to full system simulation on X86 is that it is only supported for 64-bit mode. Gem5 provides several configurations to—(i) define the underlying hardware organization (e.g., cache, memory etc.), (ii) select the level of detail required (timing, out-of-order etc.), (iii) select the debug information needed during the simulation. One of the primary advantages of Gem5 when used as a OS teaching utility is the capability to execute Gem5 in debug mode and attach debuggers like GDB [3] to correlate the underlying architectural events/states with the OS execution flow. Another advantage is the flexibility to change the micro-architectural implementation at the finest possible granularity. For example, if we want to change the semantic of a particular bit in a page table entry, it is very easily possible (changing a few lines of code) in Gem5.

[1] https://github.com/debiskms/gemOS.git

2.2 An ideal OS for teaching HW-SW co-design

Alternate approaches to address the requirements of OS and architecture co-design platforms are presented in Table 1. In the current state of Gem5, Linux is the only OS supported by Gem5. Teaching OSes like xv6, pintos etc. can not be used with Gem5 in the vanilla form. Using emulators like Qemu [11] along with the teaching OSes satisfies the objectives of a OS course to a large extent. However, modifying the architecture to the finer detail for learning and exploration is not possible because of the following reasons. First, the design objective of emulators is to provide a cross-architecture execution platform for the OSes and applications. The recent usage and focus of Qemu has been to emulate I/O devices more efficiently as it is used by many open source hypervisors. Second, Qemu uses native execution whenever possible, especially when executing a virtual machine (VM) for an architecture same as the underlying physical architecture. As a result, there is a limited scope for micro-architectural customizations.

One of the biggest challenges of using Gem5 in full system mode with Linux kernel is the sluggish execution. For example, in our experience with booting minimal Linux kernel (suggested by the Gem5 community) on Gem5 takes between eight to ten minutes of time. In order to avoid long boot time, checkpoint feature of Gem5 can be used. The checkpoint feature is useful when executing multiple application benchmarks on a given setting. For example, a checkpoint can be created after the OS boot to avoid the delays associated with booting the simulated OS. However, in the scope of this work, checkpoints does not help as the OS itself is required to be modified numerous times. Therefore, we need a lightweight OS with inherent *multi-stage execution support*. In general, a kernel with quick boot time is required for our purpose.

Another challenge of using Linux kernel as the teaching OS is the complexity of the kernel. A full-fledged OS is required to implement a variety of user APIs (e.g., system calls, sysfs and procfs), a lot of alternate policies, a wide variety of device drivers along with the architecture interfacing code. Linux provides 320 system calls even though there are a very few system calls which are used widely by a lot of applications [19]. Similarly, Linux allows the administrators to control several design parameters to control policies related to resource management, security etc. Due to the vastness of the feature set, the Linux kernel code base is really huge i.e., more than 100K LOC without device drivers and different architecture support code. It is like searching a needle in the haystack for a student or an early researcher if Linux is used as the OS for implementing lab assignments and prototypes. Therefore, a small OS that focusses more on the mechanisms, especially directly affected by the underlying architecture, rather than providing a variety of policies [10] is suitable for our purpose.

We have designed gemOS being conscious about the above requirements to keep it small and efficient (in terms of boot time); at the same time, provide enough features required to make it useful for courses focusing on OS and architecture co-design issues.

3 DESIGN AND IMPLEMENTATION OF GEMOS

High level overview of using the gemOS in Gem5 architectural simulator is shown in Figure 1. The Gem5 simulator is configured

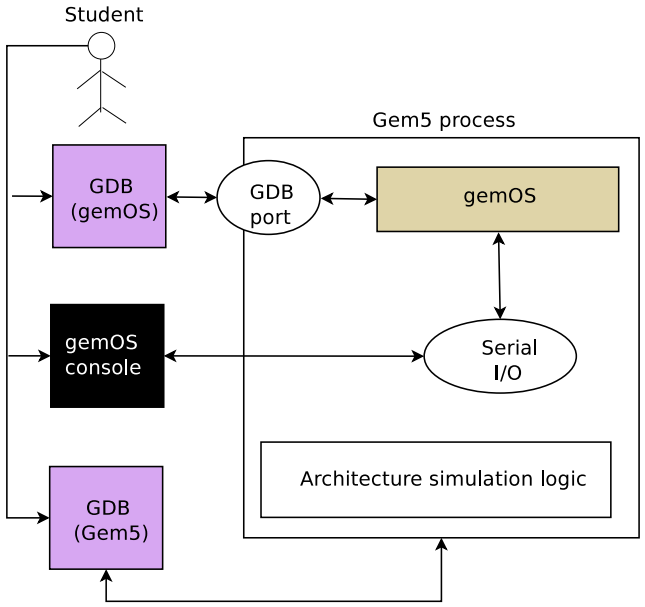

Figure 1: Overview of the proposed gemOS on Gem5 platform. The student is exposed with two separate debug interfaces—one for the gemOS and another for the Gem5 simulation process.

to run in full system simulation mode. It boots the gemOS kernel binary (from a configured path), exposes a serial terminal based I/O to the end-user through a serial port (default 3456). The gemOS kernel implements the serial device exposed by Gem5 to implement basic terminal input output. Once the gemOS kernel initializes and starts writing to the serial console, the student can see the output and sends inputs using the key board when prompted by the OS.

If the student wants to debug the gemOS kernel, she can additionally perform remote debugging with GDB using the exposed GDB port (default 7000). Just like debugging any user-level process, gemOS can be debugged and controlled from the GDB terminal. By default, the Gem5 process runs on a command line shell and prints relevant output depending on the debug settings. Just like any other process, Gem5 process can also be launched from GDB to debug the simulator along with gemOS, if required.

3.1 Multi-stage boot support of gemOS

The gemOS kernel boots into a minimal kernel-level shell as shown in Figure 2. At this point, there are no user-level contexts (processes or threads) in the system. The user can configure the OS with different configurables e.g., support for enabling global mapping feature and modified page table entry feature (See Section 4) etc. before starting the first user process i.e., `init`.

The user can start the first user process (`init`) by invoking the `init` command along with the arguments in the gemOS command line. At this point, the OS loads the `init` process and changes the current privilege level (CPL) to user i.e., set CPL = 3 before scheduling the `init` process. The code for `init` process is part of the gemOS source and compiled along with the kernel. When the

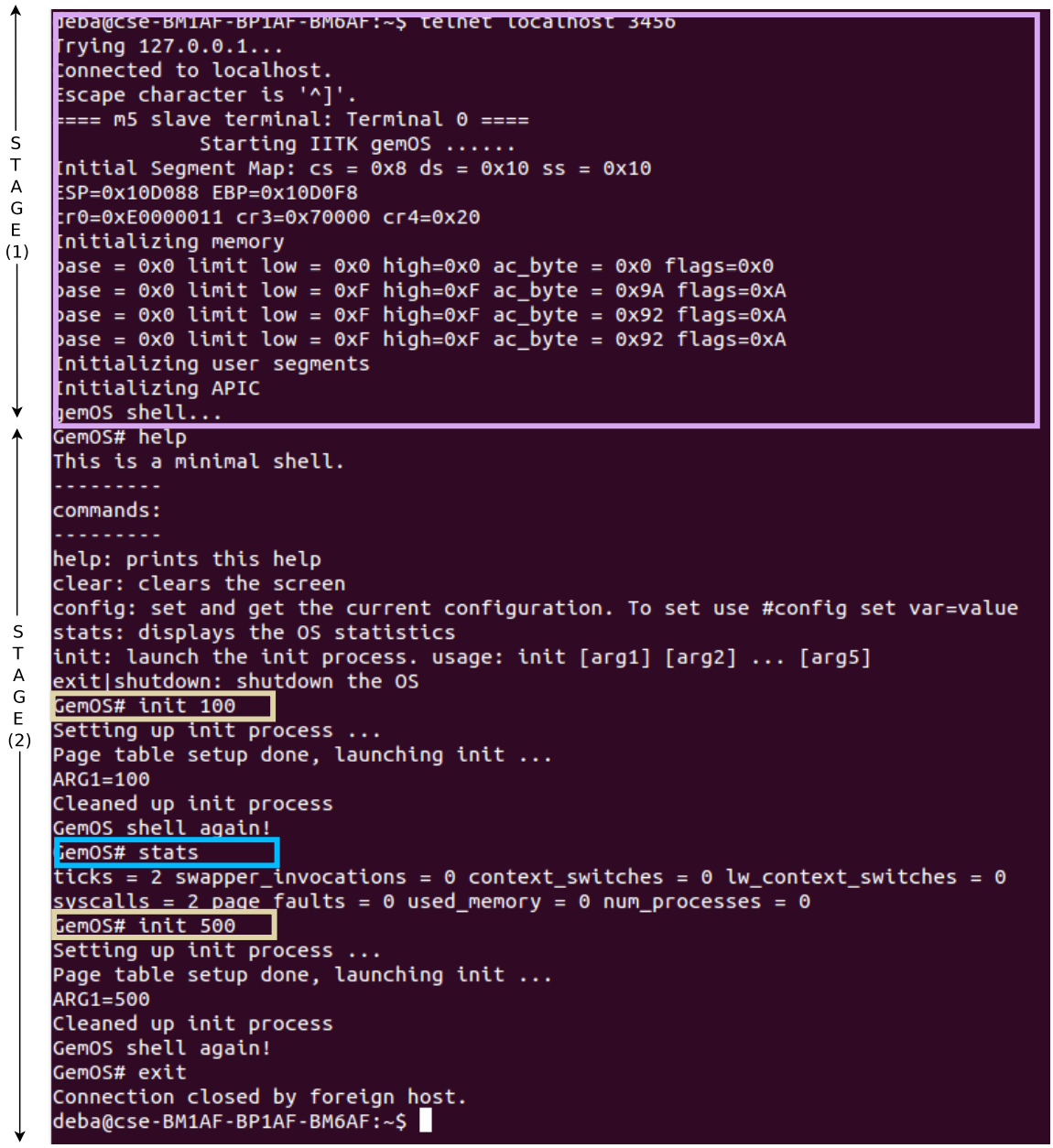

Figure 2: Multi-stage design of gemOS. Stage1: Setup the boot context (performed once) and drop into a OS-level shell. Stage2: Invoke the first user process (init) with different arguments multiple times. Statistics gathering and OS configurations can be performed from the gemOS shell.

`init` process terminates, either normally executing `exit()` system call or due to some abnormal termination (e.g., OS bug), the boot shell is invoked after cleaning up the init process. At this stage, the user can choose to exit (shut down the OS) or restart the `init` process with some other arguments.

In the example gemOS execution instance shown in Figure 2, the console output segment marked as "Stage-1" represents the first part of the OS boot before it dropped into the gemOS shell. After which the "help" command was executed to list the available commands. First invocation of the `init` process was done with one single parameter with value 100. The `init` process in this case just invokes the `write()` system call to output the passed argument value onto the console before calling the `exit()` system call. At this point, the gemOS drops onto the shell again (as there are no other user processes in this case). Different statistics related to the current and cumulative values were printed by executing the "stats" command

System Call	Explanation
`exit, getpid, fork`	POSIX compliant
`clone(stack_ptr, thread_func)`	Creates a thread which executes `thread_func` and uses `stack_ptr` for thread stack. Address space is shared and other thread semantics apply
`write (buf, length)`	Prints the contents of `buf` on console
`expand (vm_type, length)`	Expands the virtual memory area by `length` pages, physical memory is not allocated and virtual to physical mapping remain invalid
`shrink (vm_type, length)`	Shrinks the virtual memory area by `length` pages, used physical frames are reclaimed and corresponding page table entries are invalidated
`sleep(numticks)`	Calling process sleeps for `numticks` timer ticks
`signal`	Basic synchronous signal handling
`configure`	Configure different OS features
`stats`	Print the current statistics

Table 2: System call support of gemOS.

on the gemOS shell (highlighted in Figure 2). Note that, in this execution instance, there were no context switches and the number of system calls was two i.e., `write` and `exit`. The init process was re-executed with the argument value of 500 (highlighted in the figure) and was printed by the `init` process by invoking the `write` system call. Note that, the illustration presented here does not implement different behavior of `init` for different parameters (for simplicity) but is a very useful feature to execute multiple tests in a single boot.

3.2 gemOS features

The current version of gemOS supports many necessary features required for teaching a OS class and explore architectural modifications in the area of virtual memory and paging. The timer device provided by the Advanced Programmable Interrupt Controller (APIC) is used to implement a tick-based round robin scheduler, without OS-level preemption. Apart from the timer and basic console I/O, no other I/O devices are supported as we focus on the processor and memory subsystem design aspects in this work.

The system calls currently supported by gemOS are listed in Table 2. Most of the basic features like virtual memory with lazy allocation (using the page fault handler), creation of processes and threads, basic signal handling etc. are provided. For lazy allocation of memory, the physical frames are not allocated during the `expand()` system call (Table 2). Rather, the physical allocation happens on a page fault when the page is accessed by the user process.

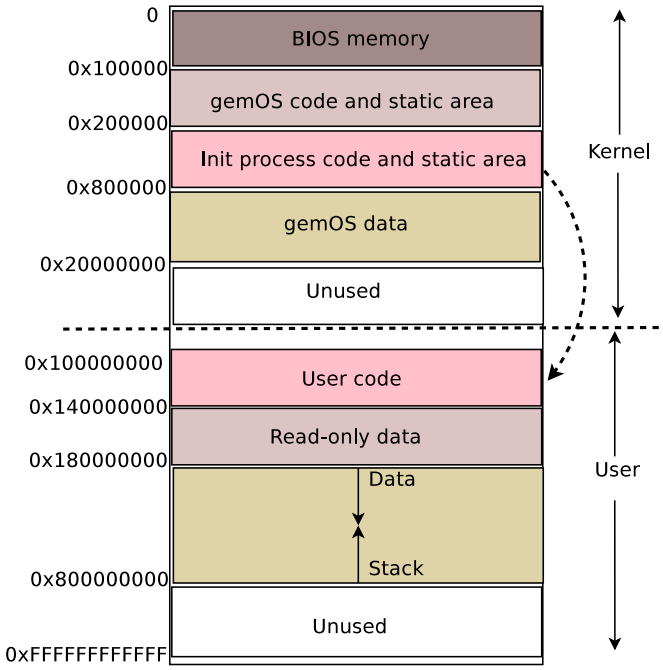

Figure 3: The process address layout of gemOS.

3.3 Implementation details

The process address space layout of gemOS is shown in Figure 3. The virtual memory subsystem of gemOS provides memory segments with different permissions, but unlike the Linux kernel, it does not provide flexible virtual memory allocation APIs like `mmap()`. The OS virtual address is canonically mapped to the physical address and is part of every user process address space. Therefore, the maximum physical memory supported by gemOS is currently 1GB but can be extended as the layout leaves a lot of gap between the user and OS addresses. To propagate the kernel mappings across all the processes, an approach similar to the Linux kernel is employed, where the top level page table entries corresponding to the kernel virtual address are propagated to the child process while handling a `fork()` system call. A chunk of the physical memory is reserved for kernel data structures and page tables and the rest is allocated to user in an on-demand fashion.

To allow entry into the OS through system calls and handle exceptions (e.g., page faults), gemOS populates several Interrupt

Feature	gemOS on Gem5
Stage-1 boot time	~ 5seconds
Stage-2 boot time	~ 30 seconds

Table 3: Time taken to boot gemOS.

Counter	Value
Ticks	2348
System calls	15
Page faults	64
Process context switches	2343
Thread context switches	0

Table 4: gemOS statistics for global mapping experiment with Tick = 1×128 bus cycles.

Descriptor Table (IDT) entries. Current implementation of system calls in gemOS is based on the old software interrupt technique (using the `int 0x80` instruction). The OS allocates one kernel stack of size 4KB for each user context (process and thread) which is used by the OS during system call or exception handling. For interrupt handling, a single global interrupt stack is used. In gemOS, process preemption in OS-level is not allowed for simplicity. However, a process can voluntarily relinquish CPU by invoking the `sleep()` system call. The current implementation of gemOS is for uni-processor simulation and is of little less than 3500 LOC.

The user space of gemOS is currently part of the OS code, albeit in a separate directory. During the OS compilation and linking, the user code is also compiled and becomes part of the gemOS kernel binary. We employ a specialized linking to create a separate section in the ELF binary for the user code starting at address 0x200000 (Figure 3). When Gem5 loads the kernel image, it also loads the user space code onto the memory address specified during the link time. To start the `init` process, we relocate the user code segment along with its static data section to the user physical memory frames and create page table translation entries to map the user code region virtual address (0x100000000) to the allocated user frames. Depending on the *init* process logic, more processes and/or threads are created using the `fork()` and/or the `clone()` system calls, respectively.

4 EVALUATION OF GEMOS

For all the experiments presented in this section, we have used the default Gem5 configuration file for full system simulation mode i.e., `configs/example/fs.py` .

4.1 Boot time of gemOS

To compare the boot time of gemOS against Linux kernel, we observed the booting time for both the boot stages. Stage-1 boot of gemOS is quite fast (Table 3) and largely depends on the number of console log events. Stage-2 consumes a little time as a lot of memory operations are required before launching the `init` process. In comparison to the Linux kernel boot time on Gem5, gemOS is efficient by an order of magnitude.

4.2 Use case: Study the benefits of global mapping

Global page table mapping support of X86 paging system allows the OS to specify some virtual addresses (e.g., the kernel virtual addresses) to be global i.e., shared across separate address spaces (processes) [16]. This feature allows the TLB to continue caching the global mappings, even after a full TLB flush by the software. Generally, a full TLB flush is triggered when CR3 (page table base register) is overwritten during a context switch. To specify global mapping, the global bit (G bit) of the corresponding page table entry has to be set to one [16]. Linux kernel used the global mapping to improve the translation performance for kernel virtual addresses prior to security attacks like Meltdown [17]. Now with security fixes like Page Table Isolation (PTI) [4], this feature is no more used in the Linux kernel. Therefore, it can be a good exercise for the students to study the benefits of global mappings.

For the purpose of this experiment, we created a new configuration in gemOS (as discussed in Section 3) to toggle between enabling global mappings and disabling global mappings. The objective of this exercise was to change the gemOS page table logic to take advantage of the global mapping feature. Additionally, basic validation of global mapping implementation and experiments to find out the utility of this feature was also part of the objective.

In order to map the kernel virtual addresses using the global bit, the page table mapping logic of gemOS for OS-level virtual address translation is required to be changed. In gemOS, this was achieved by changing just one function implementing the kernel page table mapping logic which was *less than ten lines of code*. As Gem5 already provides TLB related statistics (e.g., #of TLB lookups for read and write accesses, #of TLB misses etc.), we added some new counters (similar to existing TLB counters) to count the TLB events, specifically for kernel addresses. For example, we have added a counter `rdMissesK` to record the number of TLB misses on read access for kernel virtual addresses. The logic for maintaining the newly introduced counters was a simple additional check of CPL (= 0) in the existing Gem5 counter management code locations.

For the first experiment, we created two processes, apart from the `init` process which sleeps after creating the two child processes. Each child process allocated 128KB memory (32 pages of size 4KB) using the `expand()` system call and accessed the allocated memory for 512 iterations. In every iteration, each of the 32 memory pages were accessed four times before accessing the next page. We have used the timer interrupt interval as a parameter (configured as multiple of 128 bus cycles) for different experiments and collected the gemOS and Gem5 statistics for both with and without global mapping.

To verify the correctness, the relevant OS statistics for tick interval one (128 bus cycles) is presented in Table 4. There are 64 page faults as the first access to each memory page after `expand()` system call will result in a page fault in both the processes. 15 system calls are attributed towards creating the processes using `fork()`, some console `write()` system calls etc. Note that, there were no system call invocations during the memory access iterations. The number of process context switches were similar to the number of

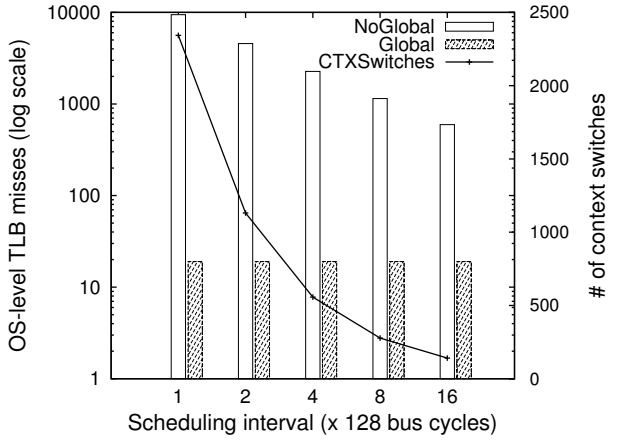

Figure 4: TLB efficiency with global mapping for OS pages with user-level working set size of 256KB.

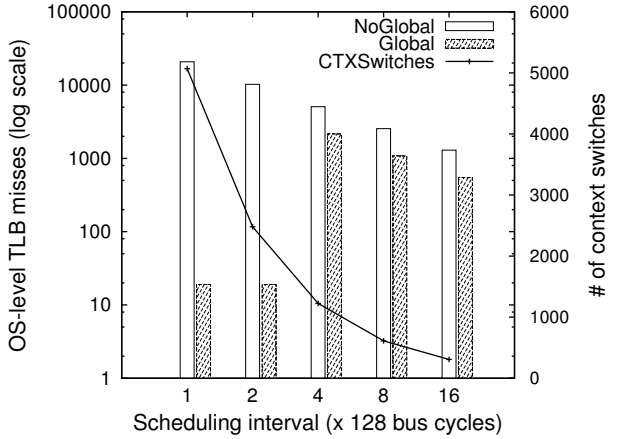

Figure 5: TLB efficiency with global mapping for OS pages with user-level working set size of 4MB.

ticks as both processes were scheduled alternatively (round robin scheduling).

We present the results of the experiment in Figure 4. The OS level TLB misses when global mapping is not used is orders of magnitude higher compared to when global mapping feature is used. Further, with high frequency timer ticks, the number of context switches is more compared to timer ticks with lower frequencies. Therefore, with no global mapping, we can observe a clear correlation between the number of context switches (presented as a line in Figure 4) and the number of OS-level TLB misses. With global mapping, the number of TLB misses for OS addresses are fixed for all the timer intervals because—(i) the OS virtual addresses are not flushed on context switch, and, (ii) the user-space working set size is 32 pages for a process, which fits in the TLB completely and does not result in eviction of TLB entries corresponding to the OS addresses.

To further analyze the impact of user memory access pattern and scheduling interval on kernel translation entries in the TLB, we

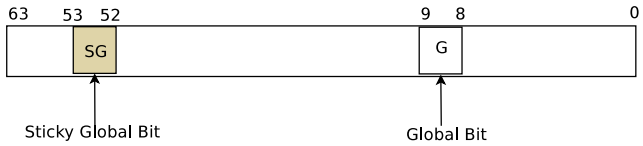

Figure 6: Modified page table entry structure to implement cross-layer sticky global mapping. The 52^{th} bit is used to represent the sticky global mappings (SG).

performed a similar experiment where both the processes allocated 2MB memory. The number of iterations are modified such that the total number of memory accesses remain the same as the previous experiment. The results for this experiment is presented in Figure 5. Interestingly, for small scheduling intervals (one and two), the number of TLB misses for kernel address translation is similar to the previous case (with 32 pages). However, for coarse grained timer ticks, the TLB misses with global paging becomes significant and comparable to without global mapping scenario. This is primarily because the timer interrupt decides the frequency of OS activity and impacts the TLB eviction logic to a certain degree. When we analyzed the TLB eviction for kernel virtual addresses, we found that certain addresses were getting evicted in a repeated manner. One of them was the virtual address mapping the APIC memory mapped I/O address, accessed in every timer tick to acknowledge the interrupt and reset the timer. In the next subsection, we present a design extension to both the architecture and the OS to realize an advanced global mapping scheme.

4.3 Use case: Extended global mapping

To avoid frequent thrashing of kernel virtual addresses in the TLB, one can change the TLB eviction logic to mark the global entries as non-evictable. However, this is a naive solution and can result in reduced TLB efficiency in many scenarios. For example, the kernel virtual addresses used during a system call at some point of time would unnecessarily occupy the TLB entries and can cause more TLB misses for the user-level virtual addresses. To make the exercise more interesting, we can propose extensions to the page table entry flags to incorporate an additional bit to specify special global mappings which are unevictable (referred to as sticky global mappings). The modified page table entry structure is shown in Figure 6.

To realize the sticky global mapping feature, both the hardware page table walker logic and the OS virtual address mapping modules are required to be modified. We believe this exercise can be suitable for advanced system courses where the students are expected to carry out research in the similar areas. To validate this proposal and find the difficulty associated with the exercise, we implemented the proposal by modifying Gem5 MMU logic and used the sticky global bit to map the APIC MMIO address (as mentioned in the previous section).

The page table logic of Gem5 is modified to add the *sticky bit* into the page table entry definition and capture the bit value during page table walk. The TLB logic is also modified to add the new bit in the TLB entry and set its value from the PTE after finishing the page table walk. We augmented the existing LRU eviction policy of the

TLB such that TLB entries with sticky global bit are not considered for eviction. A subtle pathological condition was also required to be handled when all the entries in the TLB were sticky-global. In such a scenario, the first TLB entry is evicted. All these modifications to Gem5 was a matter of adding few lines (less than 20 lines) of code. The gemOS modifications were easy as the page table entry for the kernel virtual address mapping to the APIC base address was changed to set the SG bit. After modifying both gemOS and Gem5, we repeated the previous experiment with 2MB user working set and two processes (Refer to Section 4.2, Figure 5) with timer tick set to 4. The number of TLB misses for kernel address was reduced to 1089 with the sticky global feature which was 2159 with only global bit feature.

Summary: Our evaluation shows that the proposed infrastructure (gemOS on Gem5) can be useful to explore the architecture and the OS boundaries in basic and advanced systems courses. The efforts required to realize quick prototypes of features requiring modification across the two layers depends on the nature of the exercise. The instructors can decide on the complexity of the assignments as applicable.

5 RELATED WORK

Full system architectural simulators provide implementation of the hardware components in software which can be used to implement and validate new hardware features. Gem5 [12] is the most commonly used simulator in the community. Other open source full system simulators like MARSS [18] are not as popular as Gem5. Simics [6] is another detailed architectural simulator that require commercial licence,

Teaching OSes like xv6 [7], pintos [5] were originally written for 32-bit X86 systems and meant to be used by emulators like Qemu [11] and Bochs [1]. Both the teaching OSes provide excellent documentation and debugging interfaces and some tools are also developed to improve the teaching OS on emulator infrastructure [20]. However, these OSes and setups are meant to be used for OS courses and lack the applicability into hardware interface exploration.

Cross-layer simulation tools and platforms are proposed using Gem5 as the underlying architectural simulator. For example, Hsieh et al. [15] proposed a simulation infrastructure combining Gem5 with the structural simulation toolkit (SST) to analyze HPC applications. Similarly pd-gem5 [9] simulates distributed computing systems using the Gem5 simulator. To the best of our knowledge, there are no simulation platform which allows exploration of modern-era architecture and OS in a combined fashion.

6 CONCLUSION AND FUTURE WORK

To explore the architecture and the OS boundaries for learning as well as research, Gem5 full system simulator can be very useful. We have shown that, both open source OSes like the Linux kernel and the teaching OSes are not suitable for this purpose. We proposed gemOS, a simple 64-bit X86 OS for Gem5 architectural simulator which provides a simple framework to understand the OS and architecture interactions. The academic infrastructure created by combing the Gem5 simulator and the gemOS kernel leverages the rich feature support of Gem5 and light-weight custom design of the gemOS kernel. The time taken to boot gemOS on Gem5 is shown to be very less compared to the Linux kernel boot on Gem5. We demonstrated the utility of the proposed platform towards understanding the hardware-software interactions with more clarity and implement quick prototypes to realize ideas crossing the boundaries.

As part of the future work, we intend to provide more gemOS configuration options and create a repository of exercises along with the solutions which can be reused by the academic community. Further, we also intend to add support for hard-disk device in gemOS to load complex user space logic, if required. We also plan to design visualization tools using the debug information from both Gem5 and gemOS to provide first hand experience of the OS and architecture interactions.

ACKNOWLEDGEMENT

We would like to thank all the anonymous reviewers and the members of the Computer Architecture and Operating Systems group at IIT Kanpur for their valuable comments which helped to improve the quality of the paper. This work is supported in part by IIT Kanpur initiation grant IITK/CS/2017477.

REFERENCES

[1] Bochs: The cross platform ia-32 emulator. http://bochs.sourceforge.net/.
[2] Champsim. https://github.com/ChampSim/ChampSim.
[3] Gdb: The gnu project debugger. https://www.gnu.org/software/gdb/.
[4] Linux kernel with pti. https://www.kernel.org/doc/Documentation/x86/pti.txt.
[5] pintos. https://web.stanford.edu/class/cs140/projects/pintos/pintos.pdf.
[6] Wind river simics. https://www.windriver.com/products/simics/.
[7] xv6. https://pdos.csail.mit.edu/6.828/2012/xv6.html.
[8] Adams, K., and Agesen, O. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (2006), pp. 2–13.
[9] Alian, M., Kim, D., and Sung Kim, N. Pd-gem5: Simulation infrastructure for parallel/distributed computer systems. *IEEE Computer Architecture Letters 15*, 1 (2016), 41–44.
[10] Arpaci-Dusseau, R. H., and Arpaci-Dusseau, A. C. *Operating Systems: Three Easy Pieces*, 1.00 ed. Arpaci-Dusseau Books, August 2018.
[11] Bellard, F. Qemu, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference* (2005).
[12] Binkert, N., et al. The gem5 simulator. *SIGARCH Computer Architecture News* (2011), 1–7.
[13] Christopher, W. A., Procter, S. J., and Anderson, T. E. The nachos instructional operating system. In *Proceedings of the USENIX Winter 1993 Conference* (1993).
[14] Hennessy, J. L., and Patterson, D. A. A new golden age for computer architecture. *Communications ACM 62*, 2 (2019), 48–60.
[15] Hsieh, M., Pedretti, K., Meng, J., Coskun, A., Levenhagen, M., and Rodrigues, A. Sst + gem5 = a scalable simulation infrastructure for high performance computing. In *Proceedings of the 5th International ICST Conference on Simulation Tools and Techniques* (2012), pp. 196–201.
[16] Intel. IntelÂő 64 and ia-32 architectures developer's manual: Vol. 3b. www.intel.com, 2019.
[17] Lipp, M., and Others. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Conference on Security Symposium* (2018), pp. 973–990.
[18] Patel, A., Afram, F., Chen, S., and Ghose, K. Marss: A full system simulator for multicore x86 cpus. In *Proceedings of the 48th Design Automation Conference* (2011), pp. 1050–1055.
[19] Tsai, C.-C., Jain, B., Abdul, N. A., and Porter, D. E. A study of modern linux api usage and compatibility: What to support when you're supporting. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), EuroSys '16, pp. 16:1–16:16.
[20] Walter, M., and Karlsson, S. Software tools for low-level software and operating systems classes. In *Proceedings of the 19th Workshop on Computer Architecture Education* (2017), pp. 16–23.