Lens: Experiencing Multi-level Page Tables at Close Quarters

Arun KP
kparun@cse.iitk.ac.in
Indian Institute of Technology
Kanpur, India

Rohit Singh
rsingh@cse.iitk.ac.in
Indian Institute of Technology
Kanpur, India

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

ABSTRACT

Practical understanding of the working of different sub-systems of the operating system (OS) is crucial to develop a comprehensive understanding of computing systems. Virtual memory mechanisms such as virtual to physical memory translation using multi-level page tables are commonly found in most computing devices, ranging from hand-held devices to desktops and servers. From the pedagogy perspective, one of the primary challenges in explaining virtual to physical address translation through practical demonstrations is the dependency of virtual memory sub-system on the underlying hardware architecture. Furthermore, complicated, diverse, and ever-evolving nature of the multi-level page table support for address translation presents non-trivial challenges for educators to enable a smooth journey for the students from conceptual understanding to hands-on programming experience.

While there are some system utilities to capture information related to virtual memory and address translation in open-source OSes such as Linux, these utilities are not designed for education and learning. In this paper, we propose an educational tool, *Lens*, to assist students in understanding the virtual memory concepts along with the process of virtual to physical address translation using multi-level page tables. *Lens* provides a simple, flexible, and intuitive interface which can be used to develop a holistic understanding of virtual to physical memory address translation using multi-level page tables by correlating execution of simple C programs with OS-level status of the multi-level page tables.

CCS CONCEPTS

• **General and reference** → *Experimentation*; • **Applied computing** → **Collaborative learning**; *Computer-managed instruction.*

KEYWORDS

Operating system, Page Tables, Teaching Tools, Address Translation

ACM Reference Format:
Arun KP, Rohit Singh, and Debadatta Mishra. 2023. Lens: Experiencing Multi-level Page Tables at Close Quarters. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Conference'17, July 2017, Washington,DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Virtual memory using paging is a crucial concept covered as part of operating system courses. Virtual memory concepts are basic building blocks used not only for OS development and enhancement [2, 12–14, 16, 23] but also for performance engineering [7, 10, 25, 35, 36], virtualization and cloud computing [3, 5, 6, 19, 31, 34], and embedded system development. Solid understanding of the address translation mechanisms can build strong foundation for students aspiring to become system engineers and system researchers. One important component of teaching virtual memory is to introduce the concept of virtual-to-physical address translation using multi-level page tables. The importance of multi-level page tables is due to its ubiquitous support in most modern architectures such as x86-64, ARM, RISC-V. General purpose OSes such as Linux, BSD, Windows, Solaris, and teaching OSes such as Xv6 [8], JOS [20], GemOS [18] implement the virtual memory abstraction using multi-level page table support provided by the underlying ISA. Therefore, practical exposure to the orchestration and usage of multi-level page tables in any OS course is very crucial and is the scope of this paper.

Typical OS courses introduce virtual memory concepts in an abstract manner introducing single level page tables followed by exploration of multi-level page table concepts through figures and examples. Excellent conceptual treatment of the concept in OS books [1, 30, 33] and other course materials (e.g., lecture slides) help the students to understand the working principles of multi-level page tables. However, in a practical course such as OS, first-hand practical experience and visualization plays a significant role which may not be possible to gain using the conceptual materials only. For example, to appreciate lazy physical memory allocation feature of an OS, the student should be able to examine and interpret the content of the page tables corresponding to an address allocated using `malloc()` before and after accessing the allocated memory. One possible approach to achieve this could be to profile the OS code with debug statements to generate required logs and examine them afterwards. However, there are several challenges with this approach. *First*, the OS code for paging is complex and spans thousands of lines of code which makes profiling difficult, even for simplified teaching OSes. *Second*, targeted profiling requires special skills, especially in monolithic OSes because the execution flow in OS can be really confusing for students getting introduced to the OS concepts for the first time. *Third*, because of heavy inter-dependence of OS paging logic and complex architecture specifications spanning hundreds of pages in the manuals, the student may find it very difficult to even interpret the contents of the page table.

Open source OSes such as Linux provides user space entry points (through `procfs` and `sysfs`) to collect information regarding the virtual address space (e.g., using `/proc/pid/maps`) and their physical mapping (e.g., using `/proc/pid/pagemap`). These interfaces

are designed to primarily support process memory state tracking [15, 32] used for several system services such as container migration [4, 27, 28]. These interfaces are not suitable for pedagogy purposes because they do not provide a holistic view of the multi-level page table and can not be easily co-related with program execution (explained further in §2). Tools available online [9, 24, 29] attempt to provide visual representation of the working of virtual to physical address translation. However, these tools do not show the complete end-to-end picture i.e., they randomly generate certain virtual address and translate it to a physical address instead of allowing the students to write their own programs and experience how instructions and data in their programs gets translated to physical addresses. Moreover, the existing tools abstract the virtual address translation process through a single-level page table and lack comprehensiveness to demonstrate the working of pages with hybrid sizes (4KB, 2MB and 1GB).

In this paper, we propose *Lens*, an educational tool designed to aid the users in understanding and experiencing the working of 4-level page tables used in real world operating system Linux on 64-bit x86 systems. The design objective of *Lens* is to further the conceptual understanding of multi-level page tables by providing control knobs operable by a *C* programmer and reporting the inner details of the page table levels in a simple and easy to understand form. For example, to observe the page table details of a stack variable (declared in function scope), the user needs to configure the *Lens* with the variable address and the line number in the *C* source. After the execution, the user can inspect the page table contents for the variable at the designated line number captured by *Lens* and presented in a simple tabular form. We demonstrate the utility of *Lens* through multiple use-case scenarios—practical understanding of virtual memory related system calls such as `mprotect`, demonstrating the lazy allocation of the physical memory and the hybrid page size usage (4KB, 2MB and 1GB). *Lens* is released as an open source tool downloadable from https://anonymous.4open.science/r/Lens-21F2.

2 SCOPE AND MOTIVATION

Typically students are exposed to the OS course after gaining knowledge in programming with widely used programming languages such as *C*. Therefore, involved OS concepts like virtual addressing and address translation should be demonstrated through simple *C* programs to provide conceptual continuity. We motivate *Lens* by demonstrating the challenges in connecting the execution of a simple *C* program to the OS-layer information related to the page tables.

```
1 int main(){
2  int size = 16;
3  char *ptr = malloc(size * sizeof(char));
4  printf("virtual addr of size variable: %p\n", &size);
5  printf("virtual addr of malloc'd memory: %p\n", ptr);
6  printf("virtual addr of main() function: %p\n", &main);
7  free(ptr);
8  return 0;
9  }
```

Listing 1: A simple C program showing different address regions in the process address space

Listing 1 shows a simple C program printing virtual addresses of different program regions i.e., a stack variable (`size`) in the stack region, a dynamically allocated variable (`ptr`) in the heap region and the address of a function (`main`) in the code region. This simple program can act as an example to introduce the concept of virtual memory and process address space layout in a practical manner. However, the virtual to physical address translation takes place in a transparent manner, and hence the students may not gain sufficient insight into the working of page tables by observing the program behavior only. It can be easily observed that two important practical aspects are required to connect the dots from the address of a program variable to its translation—understanding the architecture support for page tables and collection of the information from the OS. Given the lack of generality in the two aspects mentioned above, it is very difficult to gain this knowledge from text books and other conceptual materials.

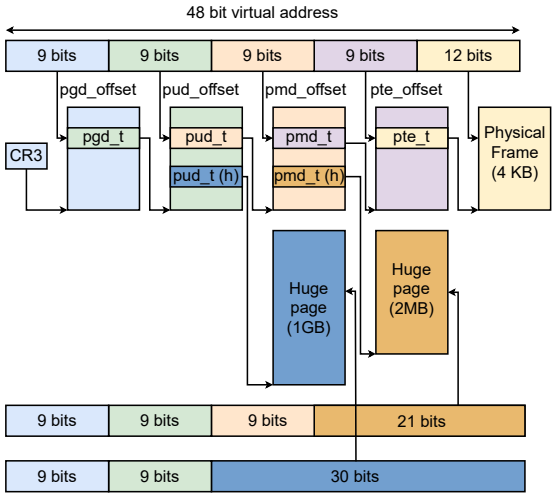


Figure 1: 4-level page table supported by Intel x86-64 processors.

2.1 Complexity of Multi-level Page Tables

Figure 1 shows the structure of a 4-level page table in the 64-bit x86 system supporting translation of 48-bit virtual addresses. As depicted in the Figure 1, the architecture specification allows the address space of a process to be constituted of a mixture of 4 KB pages, 2 MB pages and 1 GB pages. Depending on the design of OS, some of the page sizes may or may not be used. Architecture vendors such as Intel publish manuals [11] explaining the structure of the multi-level page tables, their usage and feature set in a comprehensive manner. These manuals discuss details of the topics such as address translation in great depth and cover the behavior of the translation for different combination of configurations. In our experience, even seasoned system programmers find it a daunting task to extract relevant information from the manuals.

Even after understanding the broad layout and working of 4-level page tables, one has to understand the structure and semantics of the entries at different levels to program/interpret the translation entries. For example, Figure 2 shows the structure of a page table entry in the last level page table (a.k.a. PTE) in x86-64 system. It can be observed that there are many fields present in a PTE entry

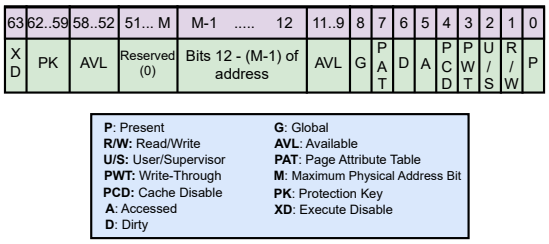


Figure 2: Page table entry in Intel x86-64 system

```
55a4c3600000-55a4c3601000 r-xp 00000000 08:02 25958718 program_code
55a4c3800000-55a4c3801000 r--p 00000000 08:02 25958718  program_code
55a4c3801000-55a4c3802000 rw-p 00001000 08:02 25958718  program_code
55a4c55fa000-55a4c561b000 rw-p 00000000 00:00 0         [heap]
7f4407200000-7f44073e7000 r-xp 00000000 08:02 36177205  libc-2.27.so
7f44073e7000-7f44075e7000 ---p 001e7000 08:02 36177205  libc-2.27.so
7f44075e7000-7f44075eb000 r--p 001e7000 08:02 36177205  libc-2.27.so
7f44075eb000-7f44075ed000 rw-p 001eb000 08:02 36177205  libc-2.27.so
7f44075ed000-7f44075f1000 rw-p 00000000 00:00 0
7f4407600000-7f4407629000 r-xp 00000000 08:02 36177201  ld-2.27.so
7f4407829000-7f440782a000 r--p 00029000 08:02 36177201  ld-2.27.so
7f440782a000-7f440782b000 rw-p 0002a000 08:02 36177201  ld-2.27.so
7f440782b000-7f440782c000 rw-p 00000000 00:00 0
7f4407929000-7f440792b000 rw-p 00000000 00:00 0
7ffd9028a000-7ffd902ab000 rw-p 00000000 00:00 0         [stack]
7ffd9034b000-7ffd9034f000 r--p 00000000 00:00 0         [vvar]
7ffd9034f000-7ffd90351000 r-xp 00000000 00:00 0         [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0 [vsyscall]
```

Figure 3: Information about mapped memory regions of the program in Listing 1 obtained from the `/proc/pid/maps` interface

whose functionality and purpose can be hard for the students to grasp. Therefore, architecture manuals designed for completeness may be too advanced for the students at the introductory level and a simple expression of the information expanding the conceptual aspects is required.

2.2 Complexity of Existing Interfaces

Widely used open source OS such as Linux provides interfaces to examine the process address space and the translation information through different user space interfaces. To find out the address translation information of a program variable (Listing 1), one has to find out the virtual address and its mapping information.

The `/proc/pid/maps` is a pseudo file maintained under *proc* file system for each process which can be used to read the virtual memory layout of a process during its lifetime. This interface provides information about currently mapped memory regions, their access permissions, and shared/private mapping type. It also provides file information for VM areas backed with files. Figure 3 shows the contents of the *maps* file corresponding to the *C* program shown in Listing 1. As shown in the figure, the output contains memory regions as ranges of addresses with many different fields. It may not be straight-forward to co-relate against a given program variable from this output. Further, the `/proc/pid/maps` interface provides limited information in terms of virtual to physical address translation. For example, it only shows read/write/execute permissions corresponding each memory region and does not exhibit information related to physical mapping.

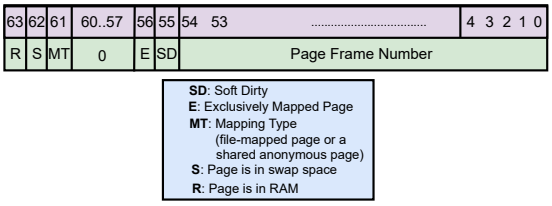


Figure 4: Layout of a eight byte entry in the `/proc/pid/pagemap` corresponding to a virtual page

Interestingly, Linux also provides another user space interface (`/proc/pid/pagemap`, referred to as `pagemap`) to examine the physical mapping information.

The `pagemap` interface is provided as psuedo file system entry in the `proc` file system for each process. The `pagemap` file can be read at a calculated offset for a virtual page to return an eight-byte entry containing information such as the page frame number allocated to the virtual page and other information related to address mapping type and permissions. Figure 4 shows the layout of an eight-bytes entry read from the `pagemap` file corresponding a virtual address. To use the `pagemap` interface in this context require modifying the program logic by manually introducing code to calculate the file offset corresponding to a program variable and collecting the mapping information using `read` system call. Further, the `pagemap` interface is designed to track virtual address mapping and writes to different memory regions using features such as `Softdirty` [15]. Therefore, the `pagemap` interface gives only the information present in the last level of a page table (PTE entry) and it doesn't allow a user to visualize the conversion of a virtual address to a physical address through a multi-level page table. Moreover, it does not provide important mapping information for huge pages and important status of the mapping (e.g., accessed, execute and other permissions). Therefore, using the `pagemap` interface, one can not experience the impact of system calls like `mprotect` and `madvise`.

2.3 Related Works

2.3.1 Teaching operating systems. Teaching operating systems like Xv6 [8], JOS [20], GemOS [18] etc. do not provide any special support to analyze the virtual memory of a userspace program. Interested users have to modify the code of these operating systems themselves to understand the working of multi-level page tables. *Lens* tool is specially designed to gain better understanding of virtual memory internals. Any user with a little bit of programming experience can use the *Lens* tool.

2.3.2 Virtual memory simulators. Virtual memory visualization (VMV) [21], PARACACHE [24], CAMERA [22] are visualization simulators for virtual memory. They allow the users to visualize the translation of a synthetic virtual memory address to its corresponding physical address using TLB and page table. They simulate a single level page table. *Lens* allows users to visualize the translation of virtual addresses of real userspace programs running on the Linux operating system to physical addresses with multi-level page table support.

An educational game has been developed by de Souza et al. [9], challenging users to convert a virtual address to a physical address in the game. This game abstracts out a lot of details of the address translation. *Lens* maintains a balance between abstraction of information as well as bogging down a user by divulging too much information. For example, apart from frame number, PTE also maintains certain flags that control how translation happens. *Lens* equips users to view the physical frame number as well as the values of these flags (such as read/write permission flag, user/supervisor permission flag etc.) maintained for each entry present in each level of the page table.

MOSS Memory Management Simulator [29], SIME [17] are some other virtual memory simulators that allow users to visualize virtual to physical address translation. However, they also have limited functionality and are not suitable for exploring the working of multi-level page tables.

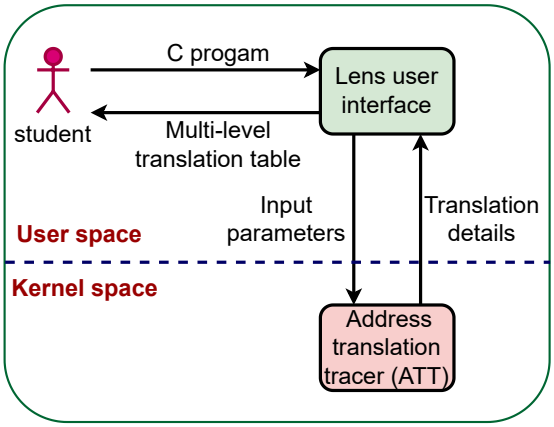


Figure 5: Schematic diagram of Lens

2.4 Motivation

Existing tools, books, manuals and OS interfaces fall short in terms of providing a simple and complete picture of the working of address translation using multi-level page tables. For clarity on this topic, the teaching team either limits the content to abstract and simplify conceptual explanation or the learners are tasked to parse complex documentation and code base to gain an up-close view. Therefore, there is a requirement for an educational tool that provides an easy understanding of the concept of address translation using multi-level page tables through visualizations. Such a tool should be easy to use and the students should be able to use this tool if they posses working background in programming. Moreover, this tool should be able to provide visual demonstration to connect programs execution with virtual to physical address translations process in real world systems. While simplicity of the interface is important, this educational tool should not compromise on the learning aspect by omitting important subtle aspects. Thus, the proposed tool should show a step by step walk through of address translation at each level in a multi-level page tables. Moreover, this tool should expose the learners to the associated concepts such as the notion of page faults, presence of huge pages in the process address space, lazy allocation of physical memory. The proposed tool (*Lens*) tries to meet these requirements as we explain in the next sections.

3 DESIGN AND IMPLEMENTATION

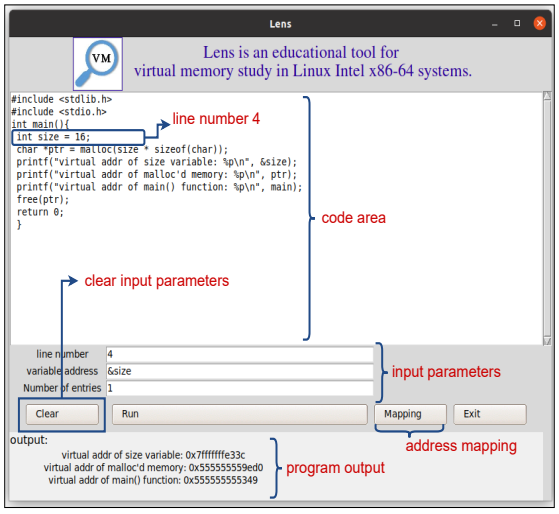
Lens is designed to enable OS learners to inspect the virtual to physical memory translation of programs written in *C* language. Figure 5 shows a high-level schematic diagram of *Lens*. It consists of an easy-to-use graphical interface at the front-end and a Linux kernel interface at the back-end (referred to as ATT) for collecting address translation details from the operating system page table.

A user uses the graphical interface to write (or copy-paste) a *C* code and provide input parameters to *Lens* (refer Figure 6a for details). *Lens* front-end coordinates with the address translation tracer (ATT) at the back-end through a kernel interface. The front end starts user code execution and provides relevant input parameters to the address trace. The tracer accesses the specific process's address translation table and captures its virtual to physical address translation details. The tracer supplies collected translation details to the *Lens* user interface for visualization and analysis. Finally, the virtual to physical address translation details of a multi-level page table is displayed on *Lens* user interface as shown in Figure 6b.

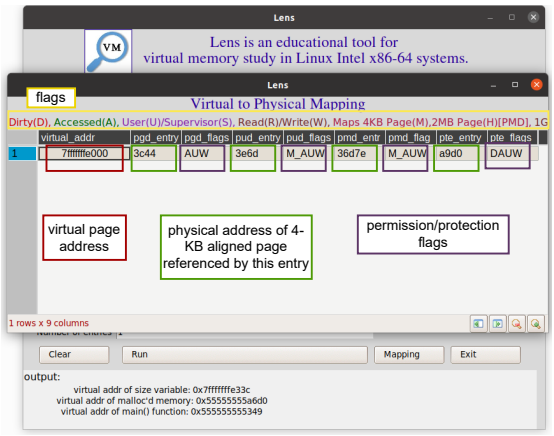
Figure 6a and 6b show the user interface of *Lens*. The user writes a C program of interest in the *code area* and provides three *Lens* input parameters—the line number, variable address and number of entries. The *line number* represents the program location where the user wants to examine the page table contents. The *variable address* parameter allows the user to specify the program variable for inspection. Using the *number of entries*, the user can specify the number of pages starting with the variable address for which the address translation information to be collected. For example, in Figure 6a, the user is interested in observing the address translation of the *size* variable. She provides the line number after the *size* variable is declared or initialized in the code, the address of size variable as *&size*, and the number of pages to be inspected as *1*. The interface shows the output of the program, and the user can navigate to the address mapping window using the *Mapping* button on the interface. Figure 6b shows the address mapping interface. The interface displays the variable's virtual page address and its translation details at different levels of a 4-level page table in Intel x86-64 systems. The details displayed in the mapping interface include the physical address of 4KB page referenced by an entry at each level and protection/permission flags currently set at each level. *Lens* also provides details about 2MB huge page mapping at PMD (page directory entry) level and 1GB page at the PUD (page directory pointer table entry) level by setting appropriate flags in the mapping interface. Users can easily modify the code and inspect corresponding address translation changes using *Lens* as we show for some example usage scenarios in Section 4

3.1 Implementation Overview

We implement *Lens* in a Linux x86-64 system with kernel version 6.1.4. Figure 7 shows the flow of events in `Lens`. We perform static code instrumentation on the input *C* program (step (1)) to include instructions to access *Lens* kernel interface and coordinate with the back-end ATT to initiate address translation tracing (step (2)).



(a) Code and input parameters interface



(b) Address translation visualization interface

Figure 6: Lens graphical user interface

The tracer (implemented as a kernel module) starts the page table walk to record address translation details using the input details passed from the user interface (step (3)). The user input includes the code line number containing the variable of interest, the address of the variable, and the number of translation entries to record, as shown in Figure 6a. The address tracer collects and stores data in a temporary staging area to be consumed later through the user interface (step (4)). The user subsequently views the address translation details about the variable of their interest (step (7)) using the visualization interface (as shown in Figure 6b) by initiating a request to fetch data from the tracer's staging area (step (5)) through the kernel interface (step (6)).

Lens's graphical user interface is implemented using Python Tkinter version 8.6 and back-end address translation tracer (ATT) is implemented as a loadable kernel module (LKM) in Linux kernel version 6.1.4.

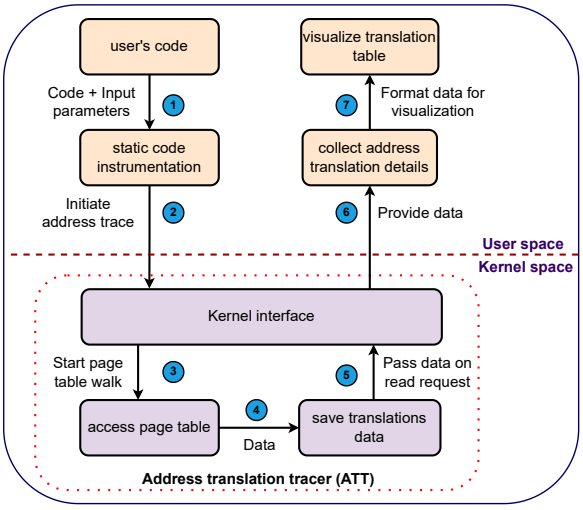


Figure 7: Flow of events in Lens

4 RESULTS

We demonstrate three use case examples to demonstrate *Lens*'s utility as an educational tool to visualize address translation using simple C programs. These use cases exhibit three fundamental OS concepts related to lazy allocation, access permission change, and huge page allocation, showcasing the clarity that *Lens*'s visualization brings to these concepts, allowing the learners to understand these concepts more effectively.

4.1 Use case 1: Lazy page allocation

The Linux systems performs lazy allocation of physical pages for user allocated virtual addresses, i.e., delaying physical page allocation until first access to the virtual address takes place. For example, in program 2, only one physical page is assigned at line number #4 even though 2MB virtual memory is allocated using the `mmap()` system call. Users can use *Lens* to visualize address translation after *ptr[0] = 'A'* to understand the fact that only one physical page is allocated at that location in the code. In program 2, students can also use the `malloc` library call in place of `mmap()` as `malloc()` uses the `mmap()` system call to serve memory allocation requests.

```
1 #define TWOMB 2097152
2 int main(int argc, char* argv[]){
3   char * ptr = (char*)mmap(NULL, TWOMB, PROT_READ|
      PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,0,0);
4   ptr[0] = 'A';
5   printf("ptr:%p\n",ptr);
6   ptr[4096] = 'B';
7   mprotect(&ptr[4096],4096,PROT_READ);
8   munmap(ptr, TWOMB);
9   return 0;}
```

Listing 2: Program used to explain use cases one and two

Figure 8 shows address mapping details from *Lens* for program 2 with input parameters as line number of `ptr[0] = 'A'` in code, virtual address of ptr[0] (i.e., `&ptr[0]`) and number of entries as 2. Figure 8 demonstrates the lazy allocation behavior of program 2 as only one physical page is allocated corresponding to `ptr[0] =`

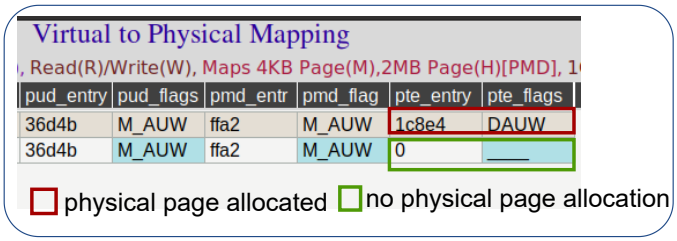


Figure 8: Output of *Lens* for program in Listing 2 demonstrating lazy allocation behavior of Linux

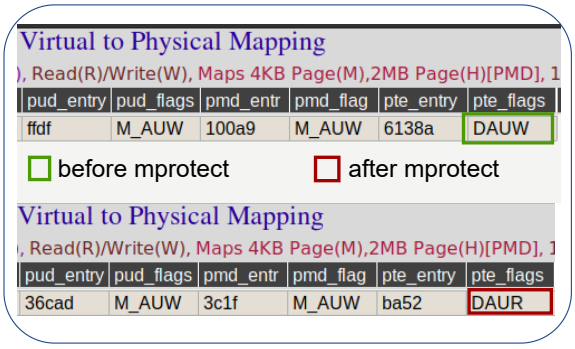


Figure 9: Output of *Lens* for program in Listing 2 demonstrating the permission change

'A' access. For the virtual address corresponding to `ptr[4096]`, no physical mapping is done as the address is never accessed.

4.2 Use case 2: mprotect() usage

Page table entry records the read(R), write(W) and execute (X) access permissions for any given page in Intel x86-64 systems. One common requirement is to change the access permission of a memory region from write to read or vice-versa. For example, to load an executable (in the form of a dynamic library), the memory region should have write permission before the load of library and the write permission should be revoked after loading the library. For this purpose, Linux provides a system call, `mprotect()`, for applications to change access permissions of an address range. *Lens* facilitates users to understand the page table manipulation of `mprotect()` like system call by visualizing permission changes in address mapping. To demonstrate this use case of *Lens*, the program in Listing 2, access permission of the second page in the mapped area is changed to read using `mprotect` at line number #7. Students can then use *Lens* to visualize address mapping of the second page (i.e., `&ptr[4096]`) before and after the execution of `mprotect()` system call.

Figure 9 shows the permission flags for the second page (i.e., the address `&ptr[4096]`) before and after execution of the `mprotect()` system call. The output clearly shows that the permission of the page is changed from write (W) to read (R) after execution of the `mprotect()` system call.

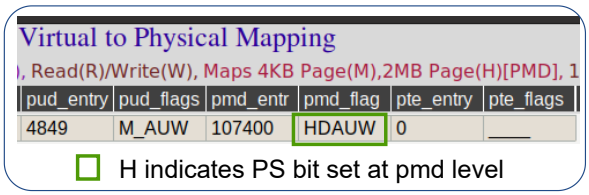


Figure 10: Address mapping from Lens for program 3 showing huge 2MB page

4.3 Use case 3: Huge page allocation

Huge pages allow better program performance by providing higher translation lookaside buffer (TLB) coverage [26]. In Intel x86-64 systems, a 2MB huge page allocation is designated by setting page size (PS) bit in PMD entry (Page-Directory Entry) [11]. Linux provides `MAP_HUGETLB` flag in the `mmap()` system call to allow users to allocate huge pages while performing memory alloaction. Linux also provides transparent huge page support [23] to promote 4KB pages to 2MB huge pages in an application transparent manner. Users can expand their understanding of huge pages by inspecting the PS bit manipulation in the PMD entry. As *Lens* provides flag details about inner page table levels, students can use *Lens* to understand the PS bit change at PMD by allocating huge pages using `mmap()` system call as shown in the program in Listing 3. In this program, we examine the content of PMD entry after allocation using the specified flags and access at line #9.

```
1  #define EIGHTMB 8388608
2  #define TWOMB 2097152
3  int main(int argc, char* argv[]){
4    int sum = 0;
5    char * ptr = (char*)mmap(NULL, EIGHTMB+TWOMB, PROT_READ
       |PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,0,0);
6    unsigned long align_ptr = (((unsigned long)ptr+TWOMB)
       >>21)<<21;
7    munmap(ptr, EIGHTMB+TWOMB);
8    ptr = (char*)mmap((void*)align_ptr, EIGHTMB, PROT_READ|
       PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS|MAP_HUGETLB
       ,-1,0);
9    memset(ptr,0,EIGHTMB);
10   munmap(ptr, EIGHTMB);
11   return 0;}
```

Listing 3: Example 3 C program

Lens denotes a huge page using the 'H' in flags output. Figure 10 shows that the `mmap` call at line #8 allocates a huge 2MB page as the 'H' flag is set in pmd_flag in address mapping information for the address `ptr`. Note that, the address need to be aligned to 2MB to allow the OS to map the virtual address as a huge page and also huge pages are enabled in Linux.

5 CONCLUSION

Practical understanding of the working of multi-level page tables is a crucial component of gaining insights into virtual memory concepts. Students aspiring to become low-level programmers should not only be exposed to clear conceptual foundation but also experience these subtle concepts through hands-on exposure. However, the complexity of the subject grows due to the hardware dependence where the architecture specifications are complicated and

OS code base for paging being non-trivial for early learners. The virtual address translation examples provided in existing educational materials are conceptual and do not capture the practical intricacies of multi-level page tables in a comprehensive manner. This leaves a gap between conceptual learning and its practical ramification in real systems resulting in lack of confidence to work on design/implementation of projects involving virtual memory sub-systems. We proposed an educational tool, *Lens*, to bridge this gap in understanding of virtual memory concepts, specifically the modern multi-level page tables, by providing an easy-to-use visualization tool to experience virtual to physical address translation using simple *C* programs.

Lens provides a simple, flexible, and easy-to-use interface for students to write C programs and correlate the program's address translation behavior with the OS-level status of the multi-level page tables. We presented three use case examples to demonstrate the ease with which *Lens* can be used to bring clarity in understanding fundamental OS concepts related to virtual memory management.

REFERENCES

[1] Peter B. Galvin Abraham Silberschatz, Greg Gagne. 2021. *Operating System Concepts*. Wiley.

[2] Andrea Arcangeli. 2010. Transparent hugepage support. In *KVM forum*, Vol. 9.

[3] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. 2010. A view of cloud computing. *Commun. ACM* 53, 4 (2010), 50–58.

[4] OpenVZ Team at Virtuozzo. 2023. Checkpoint/Restore In Userspace (CRIU). https://criu.org/Main_Page. Accessed May 11, 2023.

[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. *ACM SIGOPS operating systems review* 37, 5 (2003), 164–177.

[6] Ken Barr, Prashanth Bungale, Stephen Deasy, Viktor Gyuris, Perry Hung, Craig Newell, Harvey Tuch, and Bruno Zoppis. 2010. The VMware mobile virtualization platform: is that a hypervisor in your pocket? *ACM SIGOPS Operating Systems Review* 44, 4 (2010), 124–135.

[7] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. 2013. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel) *(ISCA '13)*. Association for Computing Machinery, New York, NY, USA, 237–248. https://doi.org/10.1145/2485922.2485943

[8] Russ Cox, M Frans Kaashoek, and Robert Morris. 2011. Xv6, a simple Unix-like teaching operating system.

[9] Darielson Araújo de Souza, Átila Rabelo Lopes, and Rosângela Marques de Albuquerque. 2015. Prototype of an educational game for teaching and learning in paged virtual memory. In *2015 International Symposium on Computers in Education (SIIE)*. 132–136. https://doi.org/10.1109/SIIE.2015.7451662

[10] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. 2014. Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 178–189. https://doi.org/10.1109/MICRO.2014.37

[11] Part Guide. 2011. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part* 2, 11 (2011).

[12] Gernot Heiser. 2020. The seL4 Microkernel–An Introduction. *The seL4 Foundation* 1 (2020).

[13] Andrew Hamilton Hunter, Chris Kennelly, Darryl Gove, Parthasarathy Ranganathan, Paul Jack Turner, and Tipp James Moseley. 2021. Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator. (2021).

[14] Robert Kaiser and Stephan Wagner. 2007. Evolution of the PikeOS microkernel. In *First International Workshop on Microkernels for Embedded Systems*, Vol. 50.

[15] Linux Kernel. 2023. Soft-Dirty PTEs. https://www.kernel.org/doc/html/v5.4/admin-guide/mm/soft-dirty.html. Accessed May 11, 2023.

[16] Jochen Liedtke. 1995. On micro-kernel construction. *ACM SIGOPS Operating Systems Review* 29, 5 (1995), 237–250.

[17] Átila Rabelo Lopes, Darielson Araújo de Souza, José Ricardo B. de Carvalho, Welk Oliveira Silva, and Verônica Lima Pimentel de Sousa. 2012. SIME: Memory simulator for the teaching of operating systems. In *2012 International Symposium on Computers in Education (SIIE)*. 1–5.

[18] Debadatta Mishra. 2019. Gemos: Bridging the gap between architecture and operating system in computer system education. In *Proceedings of the Workshop on Computer Architecture Education*. 1–8.

[19] Debadatta Mishra and Purushottam Kulkarni. 2018. A survey of memory management techniques in virtualized systems. *Computer Science Review* 29 (2018), 56–73.

[20] MIT. 2018. Josh's Operating System (JOS). https://pdos.csail.mit.edu/6.828/2018/overview.html. Accessed May 10, 2023.

[21] John Nestor and Zheping Yin. 2022. Work in Progress: A Visualization Aid for Learning Virtual Memory Concepts. In *2022 ASEE Annual Conference & Exposition*.

[22] Linda Null and Karishma Rao. 2005. CAMERA: Introducing Memory Concepts via Visualization. *SIGCSE Bull.* 37, 1 (feb 2005), 96–100. https://doi.org/10.1145/1047124.1047389

[23] Ashish Panwar, Aravinda Prasad, and K Gopinath. 2018. Making huge pages actually useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 679–692.

[24] Aryani Paramita and K.G. Smitha. 2017. PARACACHE: Educational Simulator for Cache and Virtual Memory. In *2017 International Symposium on Educational Technology (ISET)*. 234–238. https://doi.org/10.1109/ISET.2017.60

[25] Chang Hyun Park, Ilias Vougioukas, Andreas Sandberg, and David Black-Schaffer. 2022. Every Walk's a Hit: Making Page Walks Single-Access Cache Hits. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 128–141. https://doi.org/10.1145/3503222.3507718

[26] Chang Hyun Park, Ilias Vougioukas, Andreas Sandberg, and David Black-Schaffer. 2022. Every walk'sa hit: making page walks single-access cache hits. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 128–141.

[27] Simon Pickartz, Niklas Eiling, Stefan Lankes, Lukas Razik, and Antonello Monti. 2016. Migrating LinuX containers using CRIU. In *High Performance Computing: ISC High Performance 2016 International Workshops, ExaComm, E-MuCoCoS, HPC-IODC, IXPUG, IWOPH, P^ 3MA, VHPC, WOPSSS, Frankfurt, Germany, June 19–23, 2016, Revised Selected Papers 31*. Springer, 674–684.

[28] Chandra Prakash, Debadatta Mishra, Purushottam Kulkarni, and Umesh Bellur. 2022. Portkey: Hypervisor-assisted container migration in nested cloud environments. In *Proceedings of the 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 3–17.

[29] Andrew S. Tanenbaum Ray Ontko, Alexander Reeder. 2001. MOSS Memory Management Simulator. http://www.ontko.com/moss/memory/install_unix.html. Accessed May 10, 2023.

[30] Andrea C. Arpaci-Dusseau Remzi H. Arpaci-Dusseau. 2023. Operating Systems: Three Easy Pieces. https://pages.cs.wisc.edu/~remzi/OSTEP/. Accessed May 10, 2023.

[31] Fernando Rodríguez-Haro, Felix Freitag, Leandro Navarro, Efraín Hernánchez-sánchez, Nicandro Farías-Mendoza, Juan Antonio Guerrero-Ibáñez, and Apolinar González-Potes. 2012. A summary of virtualization techniques. *Procedia Technology* 3 (2012), 267–272.

[32] Rohit Singh, KP Arun, and Debadatta Mishra. 2022. LDT: Lightweight Dirty Tracking of Memory Pages for x86 Systems. In *2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 85–94.

[33] Andrew S Tanenbaum. 2016. *Modern Operating Systems*. Pearson India.

[34] Brian Walters. 1999. VMware virtual platform. *Linux journal* 1999, 63es (1999), 6–es.

[35] Lulu Yao, Yongkun Li, Fan Guo, Si Wu, Yinlong Xu, and John C.S. Lui. 2023. Towards High Performance and Efficient Memory Deduplication via Mixed Pages. *IEEE Trans. Comput.* 72, 4 (2023), 926–940. https://doi.org/10.1109/TC.2022.3191742

[36] Kaiyang Zhao, Sishuai Gong, and Pedro Fonseca. 2021. On-Demand-Fork: A Microsecond Fork for Memory-Intensive and Latency-Sensitive Applications. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 540–555. https://doi.org/10.1145/3447786.3456258