Flattening the Page Tables on Linux in x86_64

Rubin Du[*]
University of Illinois at Urbana-Champaign
rd25@illinois.edu

Shanbo Zhang[*]
University of Illinois at Urbana-Champaign
shanboz2@illinois.edu

Abstract

The abstract of the project report

1 Introduction

When the computing system was first invented, the memory was limited and expensive, and relevant logic was simple, because there were only one program running on the CPU. As time went on, the memory became larger and cheaper, and multiple users might run their programs on the same computers, but the CPU could only run one program at a time. The operating system should be able to take control of the CPU and manage the memory access of the user programs. Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory and creates the illusion to users of a very large (virtual) main memory. Paging scheme partitions the physical memory into fixed-size blocks called frames, and uses a table to map each virtual page to a physical frame. The page table contains information about each page.

Originally, the page table in x86 architecture is a two-level page table, where the address field in the first level points to the starting address of the second level page table, and the address field in the second level page table points to the starting address of the physical frame. Then, in x86_64 architecture, the page table is extended to four levels. Later, it also supports 5-level page table extension. Typically, the page table is stored in memory due to its large size. For a memory access, in the worst case, the CPU needs to walk through all the levels of the page table to find the physical address. This is inefficient, and it could be the bottleneck of the system performance. Because of the popularity of machine learning, which consists of complex memory access patterns, the page table walking could be a performance bottleneck.

In this project, we implements a new page table structure called flattened page table (FPT). FPT maintains the logical behavior and interface of the traditional multi-level tree-structured page table, while enabling the selective flattening of adjacent levels to reduce the depth of the page table walk. Specifically, assuming the 5-level paging extension is not used, we support three folding modes: L4L3, L3L2, and L4L3-L2L1 folding. In each mode, the virtual address range covered by individual levels remains unchanged, but the number of levels traversed per walk is reduced by merging multiple levels into a single larger page table. The resulting structure remains logically hierarchical, but folded levels are stored as wider tables in memory, decreasing the number of required memory accesses for translation.

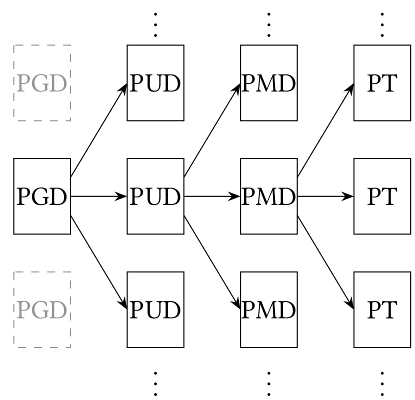

Figure 1. Traditional 4-level Radix Tree Strcuture in x86_64

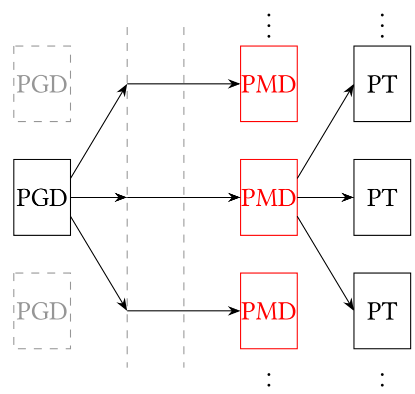

Figure 2. Flattened Page Table (FPT) in L3L2 Mode

2 Design

Figure 1 shows the traditional 4-level radix tree structure in x86_64. The radix tree is a tree data structure that stores the page table entries. Each node in the tree represents a level of the page table, and each edge represents a page table entry. The leaf nodes represent the physical frames. In the traditional 4-level radix tree structure, the uppermost level (L4) is the Page Global Directory (PGD), which points to the Page Upper Directory (PUD, L3). The PUD points to the Page Middle Directory (PMD, L2), and the PMD points to the Page Table (PT, L1). The PT points to the physical

[*]Both authors contributed equally.

frames. In our design, we enables some level(s) of tables to be flattened. The virtual address space spanned by the folded levels remains unchanged, but the size of the folded level(s) table is increased (from 4KB to 2MB). Take L3L2 folding as an example. An L3 page table page is 4KB and maps a 1GB address space, and an L2 page table page is 4KB and maps a 2MB address space. In FPT L3L2 mode, however, the level 3 no longer exists, and L2 page table page is 2MB and maps a 1GB address space. Each entry in the L2 page table page points to an L1 page table page, shown in Figure 2.

3 Implementation

In this section, we will discuss the implementation of the FPT in Linux. Since x86_64 is a complex architecture, and it is not open-source, we can only emulate the behavior of the FPT mechanism in Linux by emulators. We modified QEMU with version 6.1.50 to support the FPT mechanism in "hardware". We also used Linux kernel version 5.15.0 to support the FPT mechanism. We changed the kernel to support the FPT mechanism by modifying the logic for page allocation logic, page table walk logic, and some TLB logic.

Before we start to discuss the implementation, we need to activate the FPT mechanism in the kernel. The FPT mechanism is not enabled by default in the kernel, but allowed by some new Kconfig options - one for enabling the FPT mechanism and one for selecting the folding level, as we mentioned before. The configuration becomes effective in the kernel image after the kernel is compiled with the new Kconfig options. The kernel image is then loaded into the memory by the bootloader. The bootloader is responsible for loading the kernel image into the memory and passing control to it. We retain the backward compatibility with the radix structure, and the kernel could jump to suitable routines based on the enabled configuration.

3.1 Page Allocation Logic

The page allocation logic is responsible for allocating physical pages to the virtual address space of a process. The Linux kernel uses a buddy allocator to allocate physical pages. We do not need to modify the buddy allocator, but we need to modify the page allocation logic to support the FPT mechanism. Traditionally, the page table pages in each level have size 4KB, and the kernel uses `struct page` to represent a 4KB page frame. In FPT, however, the page table pages of the flattened level could have size 2MB. Hence, we need to modify the page allocation logic to check the level we are allocating the page table. If the level is the flattened level, we need to allocate a contiguous address space of 2MB, which corresponds to 512 (2^9) `struct page` objects.

4 Conclusion

This project is awesome.

5 Metadata

The presentation of the project can be found at:

https://zoom/cloud/link/

The code/data of the project can be found at:

https://github.com/you/repo

References