Hierarchical Segments

Yacqub Mohamed, Robbert van Renesse

1 Brief Overview

Our abstraction begins at hardware. The hardware configuration we are assuming supports multi-level page tables, huge pages, and memory protection. The abstraction we are providing over these hardware resources aims to give system developers a convenient interface to control the virtual memory layout of applications that they develop.

A tablet represents a contiguous chunk of virtual memory that is a page size the hardware supports. Each tablet is either a page table (PT) tablet or data tablet, and has a handler. PT tablet handlers do not map the PTEs inside of their tablet, and instead rent out their PTEs to other handlers, and those other handlers will map the rented PTEs to their own tablet. System developers do not create these handlers.

A segment represents a collection of tablets associated with a fixed-size region of virtual memory. An example of a segment includes the code and stack segment of a process. Each segment has a segment process, or *segproc* for short, which acts as a puppeteer over the virtual memory configuration of the segment. Segprocs will take in requests associated with virtual addresses, convert them to handler objects, and orchestrate communication between the handlers to configure the virtual memory as the system developer desires. System developers create segprocs, and the aim is to give system developers convenient control over the following characteristics:

1. Size of pages the virtual memory is mapped with (such as normal versus huge pages)
2. How pages are loaded in and where to load in their contents from
3. Which pages are revoked and where they are written back to (if written back at all)
4. How long pages are mapped for before possible reclamation by the OS

2 Motivation

Traditional virtual memory interfaces such as `mmap` provide a convenient alternative to streaming operations on files and devices. Processes can specify a memory region to be zero-initialized or associated with certain file contents, and when the memory region is accessed, the page fault handler of the operating system will initialize it. Therefore, data-driven workloads such as scientific or high-performance computing applications can map large data sets in memory, and conveniently access the data through load and store instructions.

However, `mmap` provides limited flexibility regarding the size of pages in the memory region, the page replacement policy, and whether pages are prefetched, which can hinder application performance. For example, the use of huge pages reduces page table walking latency and TLB contention, which had performance gains that motivated Google to use transparent huge pages in their memory allocator [4, 9].

`madvise` has partially alleviated this issue by allowing applications to specify access patterns for mapped memory regions. The application can specify how pages will be accessed, which pages are no longer needed, and the size of pages the region should be mapped with. This gives the kernel some information on how the virtual memory region should be set up and which pages should or should not be replaced, allowing applications to see performance benefits based on their workload-specific access patterns. But, `mmap` and `madvise` provide limited control over how the memory region is loaded in, along with where evicted pages are written back to.

`userfaultfd` is another system call in the Linux kernel meant to provide applications control over the page fault handling of their virtual memory. A fault-handling process can register a `userfaultfd` object over a mapped memory region, and poll the object for page faults. When another process attempts to access a missing or write-protected page in the registered memory region, the kernel will route the page fault to the user process polling the `userfaultfd` object. This user space customization of page fault handling has had practical use cases, such as with QEMU using missing page notifications to implement post-copy live migration, and CRIU using missing page notifications to efficiently implement checkpointing and recovery [1].

Although `userfaultfd` is highly customizable, it has shortcomings. `userfaultfd` has security issues, and is vulnerable to time-of-check-to-time-of-use (TOCTOU) attacks [5, 7]. For instance, if the fault-handling process registers a `userfaultfd` object over a memory region where no pages are present, and another process makes a system call which references that region, the kernel will block indefinitely waiting for the user space fault-handling process to load in the region. These issues have lead to Docker blacklisting `userfaultfd` by default, and Linux adding the `/proc/sys/vm/unprivileged_userfaultfd` knob, which is set to 0 by default (meaning unprivileged users can only handle user-space faults, and kernel-space faults receive `SIGBUS`). Our design aims to allow any developer to write a user-space segproc without sacrificing security or design flexibility, requiring us to completely redesign existing kernel infrastructure.

Also, `userfaultfd` gives users limited control over which pages are revoked, and where they are written back to, which is still primarily in the hands of the kernel.

`eBPF` is a promising kernel extension that allows users to dynamically load and execute user code inside the Linux kernel, subject to passing a bytecode verifier that prevents code with undesirable properties from running. `eBPF` has given applications fine-grained control over kernel managed events, such as network packet processing or file system request handling [2, 3]. User programmable memory management also seems to be possible using eBPF hooks. Users can set up virtual page `map/unmap` and `find` hooks, which the Linux kernel would reference while handling page faults. This would allow for novel paging hardware to be tested within the Linux environment [6]. eBPF has high promise, however, our design aims to rework the kernel's virtual memory abstractions from the ground up, and only allow for users to manage their memory from user-space.

3 Design

This section will first describe the hardware model we are assuming. The description of our hardware model will then be followed up by a discussion of tablets, which are an abstraction over the bare hardware resources. Then, this design section will describe segments, which are an abstraction over tablets that system developers can control. Finally, our threat model will be described, and policies will be presented (and enforced) that prevent the threat from enacting harm on systems built on top of our virtual memory abstraction.

3.1 Hardware Model

3.1.1 Multi-Level Page Tables

A frame is a 4 KB chunk of data. A page is a frame associated with a page number and metadata in the virtual address space of a process. This metadata includes whether the page is present, its access permissions, and dirty status. The page number acts as a name of a frame, and a frame can have many different names. If a frame has no name, then it is free to allocate to any process when necessary.

A page table translates page numbers to frame numbers. Processes issue load and store requests to access bytes in pages using an *address*, which is a (page number, offset) pair. The MMU of a processor will split a page number into a sequence of N equal-length indices, where N represents the number of levels in the page table. The indices represent a path in the page table hierarchy, and the MMU will walk down this path to find the frame number associated with the page number. If the page is not present or the access is disallowed, the MMU will generate a page fault.

3.1.2 Huge Pages

A huge page is a contiguous collection of frames associated with a page number and metadata in the virtual address space of a process. The metadata of a huge page is identical to the metadata of a standard page, but a huge page is usually 2 MB or 1 GB in size, instead of the standard page size of 4 KB. Huge pages are also referenced using addresses, but the MMU will interpret an address which references a huge page differently. An address will still be partitioned into a (page number, offset) pair, but the number of bits in the offset will be larger, as a larger page size requires a larger offset to fully index the page. Therefore, the page number is smaller, implying the number of indices in a page number is smaller, so references to huge pages require fewer memory accesses to translate.

For example, in a two level page table scheme with 32-bit addresses, a standard page size is 4KB, meaning the page number is 20 bits. The MMU will split the page number into two 10 bit indices, which represents a path in the two level page table. Now, a huge page is 4 MB, meaning the offset for it would be 22 bits, and the page number would be 10 bits. Therefore, the MMU only needs to access the root page table to translate a page number referencing a huge page.

Although the MMU in most modern processors support 1 GB huge pages, it is difficult for the OS to find a contiguous sequence of frames in physical memory that amounts to 1 GB. Recent research has attempted to alleviate this

problem by introducing dynamically adjustable regions of contiguous memory in the Linux kernel, creating more opportunities for large allocations [8]. Another possibility is that the OS can use memory protection mechanisms to allocate the frames on demand. For a simple example, suppose a huge page was 8 KB, but the OS could not find two consecutive frames to allocate for it. The OS can allocate one frame, map the huge page to start at the allocated frame, and restrict access to the frame directly after the allocated frame. Whenever the second (restricted) frame is accessed, the OS can then migrate the contents of the restricted frame elsewhere, update the restricted frame's contents, and remove the memory protection on it.

3.2 Tablet

A tablet represents a contiguous range of virtual memory that is a hardware-supported page size. The tablet's size can be represented by the number of bits needed to index every byte in the virtual range. A tablet containing 2^b bytes has a size of b bits. A tablet can be a page table (PT) tablet, meaning the tablet is associated with single page table frame, and contains a collection of PTEs. A tablet can also be a data tablet, meaning the tablet is associated with a contiguous sequence of frames, and each of the frames can contain arbitrary contents.

A tablet has a single owner, denoted as *handler*, which is a first-class object associated with a handler identifier (HID), much like a process is uniquely identified by a process identifier (PID). Handlers for PT tablets should not be able to point PTEs to arbitrary places. Therefore, updates to a PT tablet are authorized by the kernel.

Handlers for PT tablets create *capabilities* for PTEs inside of their tablet, denoted as a *rental*, and shares these rentals with other handlers. Other handlers can only map authentic rentals to their own tablet, subject to constraints defined by the OS along with the handler which created the capability.

3.2.1 Capabilities

A capability is a token that represents privileges for an object. Having access to the token means having certain access to an object. This provides a workaround to using authentication schemes. The token must be tamper proof and unforgeable to meaningfully enforce an access control policy. There are two types of capabilities.

The first is **cryptographically protected** capabilities, which rely on cryptographic tricks to retain authenticity. A cryptographically protected capability can be a large random number, which is hard to guess. It can also be encrypted to maintain confidentiality properties, or signed to maintain integrity properties.

The second is **kernel protected** capabilities, which rely on the kernel to retain authenticity. The kernel can maintain *capability sets* for processes, and when a process attempts to access an object with a capability, the kernel is invoked and will ensure the capability is in the capability set of the process.

Our design uses cryptographically protected capabilities. Identifiers for first-class objects are given large random numbers to maintain their confidentiality, and rentals are signed using the kernel's signing key to maintain their integrity.

3.2.2 Handler Primitives

A PT tablet handler is allowed to invoke all three primitives below, whereas a data tablet handler is only allowed to invoke the latter two.

1. $\texttt{rent}(\texttt{index}, \texttt{perms}, \texttt{duration}) \rightarrow \texttt{rental}$: The caller (PT handler) can rent out one of its PTEs for a fixed amount of time[1]. The index argument is an index into the tablet the caller manages, which locates a PTE, and the kernel will take the following actions to create the rental

   $$\texttt{m} := (\texttt{version}, \texttt{frame}, \texttt{index}, \texttt{perms}, \texttt{expTime})$$

   The `version` element is the version number associated with the tablet. Every time the tablet is allocated, it has its version number incremented. This argument is meant to prevent old rentals for tablets from being valid if the tablet were freed and subsequently reallocated. The `frame` element is the frame number of the caller's tablet (since a page table tablet is uniquely associated with a single frame number).

   $$\texttt{rental} := (\texttt{m}, \texttt{sign}(\texttt{k}_{\texttt{os}}, \texttt{m}))$$

   The OS will sign off on the rental with its signing key ($\texttt{k}_{\texttt{OS}}$) to maintain the rental's integrity. The rental is returned back to the caller, and it is up to the caller to disseminate this rental to the desired handler so that they can map the PTE to their tablet.

   To enforce the expiration time constraint, the OS maintains a log of active finite rentals. The OS will routinely search this log, and will forcefully unmap expired rentals.

[1] The `perms/duration` argument for a PTE in a non-leaf PT tablet could specify RWX permissions/finite duration, since the PTE can map a huge page. If the PTE is not being mapped to a data tablet, the perms must be `NULL` and duration must be infinite.

2. `map(rental)` : The OS will map the PTE specified in the rental to the caller's tablet (using the permissions in the rental)[2].

   The OS will check four things before creating the mapping. Firstly, the OS will use its verification key to verify the integrity of the rental. Secondly, the OS will check that the rental has not yet expired. Thirdly, the OS will check that the PTE referenced by the rental is not already being used.

   Finally, the kernel will enforce a size restriction on the tablets involved with the mapping. A tablet of size b can only be mapped by a PTE inside a tablet of size $b+\Delta$, where Δ is the size of a page number index. The reason for this restriction is to retain the size of the virtual range a tablet manages. A tablet of size $\Delta+b$ bits contains 2^{Δ} PTEs, each of which map a virtual range of size 2^b.

3. `unmap(rental)` : WIP

3.3 Segment

A segment is a fixed-size region of virtual memory associated with a collection of tablets. A segment is named by a *segment number*, which is an index into the root page table of a process. Thus, a page number can be split into a (segment number, segment index) pair, where the *segment index* represents a page inside of the segment. That means a process can have up to 2^{Δ} segments ($\Delta =$ page number index size), where each segment is rooted at its own page table tree (PTT) mapped by a PTE in the root page table of every process that maps the segment.

A segment is managed by a *segproc*, which is also a first-class object identified by a segment identifier (SID). Segprocs own a segment, but do not own any of the tablets inside of its segment. A segproc also does not map/unmap any PTEs inside of tablets, and instead develops communication protocols with handlers inside of its segment to modify the the segment's virtual memory layout. When a segproc is invoked to handle a page fault, the segproc is given the name of the page where the fault occurred (segment index), and converts this name into a sequence of handler objects to contact.

The main functions of segprocs are to handle page fault and revocation events. System developers create these event handlers, and are given basic primitives to allocate tablets, locate tablets using their virtual names, load in virtual memory contents, as well as free tablets.

[2]The `perms` in the rental have to be `NULL` and the `expTime` must be infinite here if the caller manages a page table tablet.

3.3.1 Segproc Primitives

Segprocs are given five basic primitives to coordinate the virtual memory configuration of the segment they manage.

1. `tablet_alloc(type, size)` $\rightarrow$ `handlerID` : Allocate either a data tablet or PT tablet which represents a particular virtual memory size, and spawn a handler to manage the tablet. The caller segproc becomes the parent of the allocated tablet, and the handler identifier of the spawned handler is returned.

   The type argument determines whether the tablet is a data tablet or PT tablet. The size argument is the number of bits needed to fully offset into the virtual memory associated with the tablet, and must be a hardware-supported page size. For example, a 2 MB data tablet would need a size of 21 bits, and `tablet_alloc` will allocate a contiguous sequence of frames for the tablet which comprises 2 MB of physical memory. A PT tablet of size 21 bits would be a leaf PT frame (as the tablet can map up to 2^9 data frames of size 2^{12} bytes), and would only require a single frame allocated.

   A handler is spawned which "owns" the tablet. A segproc has access to a tablet if it knows the handler identifier of the handler which owns the tablet. A handler identifier is just a number, but must be carefully crafted so that it cannot be easily guessed. But, if a handler identifier cannot be easily guessed, it is likely hard to remember, and won't give any hints about which page(s) in the segment are managed by the handler. Therefore, the segprocs need a directory service to convert virtual names of handlers to their handler identifiers (physical).

2. `tablet_locate(name)` $\rightarrow$ `tablet_info` : Find and return the metadata information associated with the tablet that most closely matches the `name` argument.

   The name of a tablet is a *(segment index, size)* pair, and is interpreted in the context of the caller's segment. The segment index determines the address where the tablet's virtual memory starts, and the size is a bit length meant to determine the range of virtual memory the tablet manages.

   The kernel will begin a page table walk starting from the PTT where the caller's segment is mapped at. The *segment index* in the `name` argument defines a path in the PTT, and the tablet metadata where the walk stopped (size match or PTE invalid) is returned.

3. `tablet_map(handlerID)` → `address` : The kernel will map the tablet owned by the handler identified by `handlerID` in the address space of the calling segproc. The address where the tablet was mapped is returned.

   If the handler specified in the argument is a data tablet, the kernel will map the frame(s) associated with it as RW. Otherwise, the handler is for a page table frame, and the kernel will map the frame as RO. The kernel will then return the base address where the frame(s) are mapped (which is aligned based on the tablet's size).

4. `tablet_unmap(address)` : Inverts `tablet_map` (using an address returned by `tablet_map`).

5. `tablet_delete(handlerID)` : Terminate the handler identified by its handler identifier, and free all the frames associated with the tablet the handler owns. A tablet can only be deleted if it has a reference count of zero.

3.3.2 Segproc Event Handlers

A segproc is defined by four event handlers, and system developers must create these event handlers when defining a segproc. These event handlers can use the primitives described above. When an event defined below occurs inside of a segment, the proper event handler of the segproc managing the segment is invoked. For notation, "you" refers to the system developer creating the segproc.

1. `init` : Invoked when the segment is created by the OS. You should pre-allocate tablets and map the virtual memory of the segment as you desire.

2. `pagefault` : Invoked when a process experiences a page fault inside of your segment. You are given the segment index of the fault and type of access. It is up to you to locate the handler objects associated with the fault, allocate new tablets, load them in, and coordinate mappings to resolve the page fault.

3. `reclaim` : WIP

4. `unreachable` : Invoked when a tablet you are a parent of hits a reference count of zero. The parent is defined by the segproc which allocated the tablet, and if the segment dies before its child tablet, an arbitrary segment that maps the tablet becomes the parent (or the tablet is reclaimed if no other segment maps it).

   You are given the `tablet_info` of the unreachable tablet, and can either delete it or attempt to have it mapped again. A tablet that has a reference count of zero for too long will be forcefully reclaimed by the OS.

Figure 1 is an example of a page fault event handler that a system developer may write. The tablet interfaces allow for system developers to cleanly manage the virtual memory configuration of the segment without worrying about low-level hardware specific details (such as the format of PTEs). System developers do, however, have to know what page sizes are supported by the hardware their programs are running on.

3.4 Threat Model and Policy

The threat in our system is a segproc, which in turn contains the motivation of the system developer which wrote it. To understand what policies are necessary, we must first understand what the segprocs are capable of, and what their motivation is.

A segproc can use standard message passing routines provided by the kernel. As a result, a segproc can send messages to any handler/segproc, provided the sender knows of the receiver's unique identifier. A segproc can also only read messages that are sent to them. We are assuming the only way a segproc can learn of the unique identifier associated with a first-class object is by random guessing, or through the contents of a received message. Finally, segprocs can attempt to forge rentals and tamper with rentals they have received [3].

One possible motivation a segproc may have is to break the isolation provided by the OS, in effect to learn the contents of sensitive data that another process holds in its memory. For example, there may be an encryption application which holds unencrypted messages on its stack. One possible harm a segproc can enact is to forge a rental for a leaf PTE inside the stack segment of the encryption application, and map the forged rental inside of its own segment. Then, if the encryption service begins writing unencrypted messages into a buffer on its stack, the malicious segproc could read the contents of the messages. Another harm could be that a malicious segproc guesses the handler identifier of a data tablet inside of the encryption service's stack segment, and messages the handler to map a rental the segproc generated. The malicious segproc can then read the unencrypted messages.

[3] A segproc could also neglect to handle a page fault, but this would only impact the availability of processes which mapped the segment. Our kernel design pins a page for each process to place system call arguments in, meaning the TOCTOU vulnerability is avoided with a slight memory consumption cost.

```
pagefault(segidx, type):

    # allocate data tablet and load it in
    handler_child = tablet_alloc(TABLET_DATA, PAGE_NBITS)

    address = tablet_map(handler_child)
    #
    # load in page worth of content from any storage into [address]
    # (or zero initialize it)
    #
    tablet_unmap(address)

    # locate leaf page table tablet, obtain indefinite rental
    # from its handler, and get data tablet handler to map the rental
    leaf_tinfo = tablet_locate(name=(segidx, PAGE_IDX_NBITS+PAGE_OFF_NBITS))

    send(leaf_tinfo.HID, RENTAL_OBTAIN=(indefinite), PTE=last_index(segidx))
    recv(leaf_tinfo.HID, rental)

    send(handler_child.HID, RENTAL_MAP, rental)
    recv(handler_child.HID, status)

```

Figure 1: Example page fault event handler inside of a segproc. When a segproc is invoked with a page fault event, the segproc is given the segment index where the fault occurred and the type of fault. This code assumes the leaf page table tablet containing the fault already exists, but the faulting page itself is not present. The segproc will first allocate a data tablet and load in its contents. Then, the segproc will find the leaf PT tablet which contains the faulting page, and tell the data tablet to map a rental the leaf PT tablet handler generated.

There are two policies that our design will enforce to prevent these harms. The first is that rentals for PTEs are unforgeable and tamper-proof. This is enforced by having the OS sign rentals with its signing key. Every mapping of a rental invokes the kernel, which then checks the integrity of the rental before updating a PTE. The second is that only intended communication between segprocs and handlers should be allowed. "Intended" is loosely defined, but an example could be that a segproc can only learn of the handler identifier of tablets mapped in its segment. This is enforced by using large random numbers as the unique identifiers for handlers and segprocs, which prevents identifiers from being guessed. Also, the `tablet_locate` functionality only allows a segproc to determine the handler identifier of tablets mapped in its segment.

4 Tying it all Together

Take the case of a 48 bit virtual address, with 4 KB pages and a 9 bit page number index. Then a process contains a collection of 512 segments, where each segment maintains a 512 GB range of virtual memory. A segment is named by a *segment number*, which is an index into the root page table of a process. Each segment contains a collection of *tablets*, and is managed by a *segproc*, which orchestrates communication between the tablets inside of the segment.

A tablet is a collection of virtual memory that is a hardware-supported page size. The name of a tablet is a (segment index, size) pair inside of a segment (meaning the root page table of a process is not a tablet). Each tablet is owned by a *handler*, which rents out the PTEs inside of its tablet, and maps rented PTEs to its own tablet.

An address can be viewed as a (segment number, segment index, offset) triple, where a segment index names a page inside of a segment, and the segment number names the segment which contains the page. The offset into a page names a byte.

References

[1] CRIU Project. `https://www.criu.org`.

[2] Ashish Bijlani and Umakishore Ramachandran. Extension framework for file systems in user space. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 121–134, Renton, WA, July 2019. USENIX Association.

[3] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hXDP: Efficient software packet processing on FPGA NICs. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 973–990. USENIX Association, November 2020.

[4] Andrew Hamilton Hunter, Chris Kennelly, Darryl Gove, Parthasarathy Ranganathan, Paul Jack Turner, and Tipp James Moseley. Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, 2021.

[5] Jonathan Corbet. Blocking userfaultfd() kernel-fault handling. `https://lwn.net/Articles/819834`.

[6] Kaiyang Zhao. Programmable Memory Management with eBPF .

[7] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. Dirtycred: Escalating privilege in linux kernel. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 1963–1976, New York, NY, USA, 2022. Association for Computing Machinery.

[8] Kaiyang Zhao, Kaiwen Xue, Ziqi Wang, Dan Schatzberg, Leon Yang, Antonis Manousis, Johannes Weiner, Rik Van Riel, Bikash Sharma, Chunqiang Tang, and Dimitrios Skarlatos. Contiguitas: The pursuit of physical memory contiguity in datacenters. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[9] Weixi Zhu, Alan L. Cox, and Scott Rixner. A comprehensive analysis of superpage management mechanisms and policies. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'20, USA, 2020. USENIX Association.