VerifyThis 2026 Task 3: Multi-Level Page Tables

Proposed by Hans-Dieter Hiep

1. Introduction

Modern operating systems and hypervisors make use of hardware-based Memory Management Units (MMUs) that implement "virtual memory". Virtual memory, amongst other advantages, allows for memory isolation between processes, by ensuring that each process can only access the memory it has been allocated.

In particular, when a "user" program allocates memory, it is given a *virtual address*, which will be translated by the MMU to a *physical address* (an actual location in memory) when the program accesses it. This translation is performed using a data structure called a *multi-level page table*.

To do so, the MMU maintains a mapping from *virtual addresses* to *physical addresses* using a *multi-level page table*, and performs address translation on every memory access. A "user" program only sees virtual addresses.

In this task, teams will specify and verify memory management algorithms, on top of a simple MMU abstraction. While we make many simplifications, we maintain a certain level of complexity by working with complex bitvector manipulations, pointer arithmetic, and multi-level page tables. The participants are free to reimplement the algorithms in a different way, or simplify the representations of the various abstractions. However, the jury will reward solutions that are faithful to this document.

All data structures are provided, and algorithms are given in python-like pseudocode.

2. Virtual Address Translation Using Multi-Level Page Tables (Q1)

2.1. Machine and Memory

For the sake of this challenge, a *machine* consists of:

- a physical memory `mem`, which is a contiguous array of 2^{32} bytes, naturally indexed by *physical addresses* that are unsigned 32-bit values; that is, 4 GiB of memory, with addresses of type `u32`;
- an allocator state composed of two physical addresses `m_next` and `h_next` used by the page bump allocators for metadata and heap pages respectively (see Section 3).

```pseudocode
struct Machine {
    mem: byte[2**32],
    m_next: u32,
    h_next: u32
}
```

```pseudocode

type paddr = u32
type vaddr = u32

```

For simplicity, all algorithms in this task manipulate only 4-byte-wide unsigned integers, including physical addresses (`paddr`) and virtual addresses (`vaddr`). Physical memory is accessed using the functions `M.mem.read_u32(a)` and `M.mem.write_u32(a, v)`, where `a` is a physical address and `v` is any value of type `u32`.

We define a *page* of memory as a contiguous block of $2^{10} = 1024$ bytes (1 KiB) which is aligned to a page boundary (i.e., its start address is a multiple of 1024).

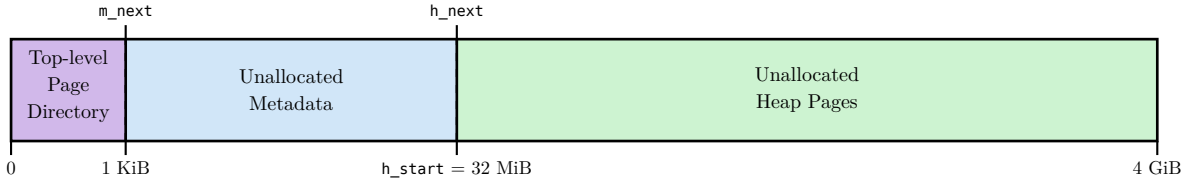

Figure 1: Physical memory layout at startup of the machine (not to scale)

Figure 1 and 2 show the layout of physical memory, and the meaning of the allocator state. Figure 1 shows the layout of the physical memory at startup of the machine. The first page of memory (the first 1 KiB) is reserved for the *top-level page directory* (explained below in Section 2.2), which is the root of the page table structure. Then, the slice of memory from 1 KiB to 32 MiB are reserved for metadata (used

by the page table structure). Finally, the rest of the memory (from 32 MiB to 4 GiB) is reserved for heap pages that will be allocated to the user. On startup, no metadata or heap pages have been allocated, so the bumping pointers `m_next` and `h_next` are both set to the start of their respective regions (1 KiB and 32 MiB). Furthermore, and this is not shown in the figure, at startup, the top-level page directory is initialized to be empty (all entries are zero). In addition, we use the constant `h_start`$= 2^{25} = 32$ MiB to denote the boundary between the metadata and heap regions.

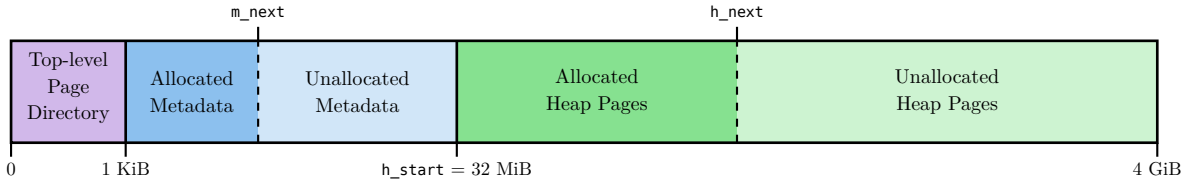

Figure 2: Physical memory layout after some allocations have been performed (not to scale).

After some allocations have been performed, the layout of physical memory is as shown in Figure 2. Up to `m_next`, metadata pages have been allocated, and up to `h_next`, heap pages have been allocated. The rest of the memory is still free.

2.2. Virtual Addresses and Page Tables

The role of the metadata region is to store *page tables* that contain the information necessary for the MMU to translate *virtual addresses* to physical addresses. Specifically, a virtual address `vaddr` is a 32-bit integer that is composed of four contiguous fields, represented in Table 1: top-level offset, mid-level offset, leaf offset and page offset.

	top-level offset						mid-level offset								leaf offset								page offset							
bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Table 1: **Virtual Address**: components

To understand the role of each field, let us first describe the anatomy of a page table. As represented in Table 2, a *page table* is a single page of memory (i.e., an array of 256 values) where each value is a 32-bit physical address aligned to a page boundary (meaning that it is a multiple of 1024). Because of this alignment, the last 10 bits of each value are guaranteed to be zero, and can be used to store additional metadata on the pointer. We use the least significant bit, denoted `P` for "Present", to encode whether the page is allocated (1) or not (0).[1] The layout of a page table entry is given in Table 3.

Address	Entry (4 bytes)
$\ell + 0$	xxxxxxxxxxxxxxxxxxxxxx000000000P
$\ell + 4$	...
⋮	
$\ell + 1020$	...

Table 2: **Page Table**: memory layout of page tables

	significant bits																						always 0								P
bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Table 3: **Page Table Entry**: components

Page tables are the data structure used by the machine to translate virtual addresses to physical addresses. Each virtual address goes through three levels of page tables: a top-level page directory, a mid-level page directory, and a leaf page table. The top-level page directory is located at the first page of the physical memory (the first 1 KiB), and is indexed by the most significant 6 bits of the virtual address. For instance, if the first 6 bits of the virtual address are all `0`s, then the first entry `PD[0] = mem[0]` of the top-level page directory is used. This entry is the address of the mid-level page directory (located in the metadata region) that must be looked up. The least significant bit of the entry, `P`, indicates whether the page of the mid-level directory is allocated; if it is not, then the translation fails. The process is then repeated for the mid-level directory, using the next 8 bits of the virtual address, to find the leaf page

[1] In a more realistic MMU, there would be more metadata bits, such as read/write permissions, user/kernel mode, etc.

table (still located in the metadata region), and then for the leaf page table to find the physical address of the heap page (located in the heap region). Finally, the last 10 bits of the virtual address are used as an offset within the heap page to find the exact physical address to a specific byte. The entire function that translates virtual addresses to physical addresses, `translate`, is given in Listing 1.

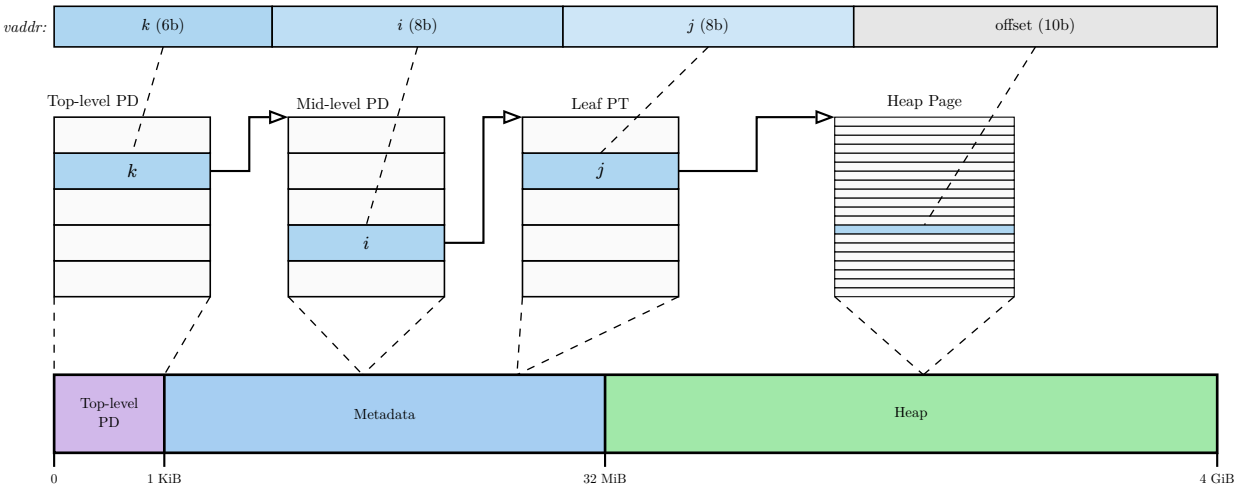

Figure 3: Virtual address translation: the three bit-fields k, i, j select values within successive levels of page tables; the offset selects a byte within the heap page.

pseudocode

```
# Returns: physical address, or None if any page table entry is absent.
def translate(M: Machine, vaddr: u32) -> u32 | None:
    k = vaddr >> 26                  # Offset within top-level directory (6 bits)
    i = (vaddr >> 18) % 256          # Offset within mid-level directory (8 bits)
    j = (vaddr >> 10)  % 256         # Offset within page table (8 bits)
    o = vaddr % 1024                 # Offset within page (10 bits)

    # The top-level directory is located at physical address 0.
    t = M.mem.read_u32(k*4)          # "* 4" because each entry is 4 bytes
    if t % 2 == 0: return None       # Page is absent
    t = (t >> 10) << 10              # clear low 10 bits (flags)
    # t is now the physical address of a mid-level directory

    m = M.mem.read_u32(t + i*4)      # read mid-level directory entry
    if m % 2 == 0: return None       # Page is absent
    m = (m >> 10) << 10              # clear low 10 bits (flags)
    # m is now the physical address of a page table

    p = M.mem.read_u32(m + j*4)      # read page table entry
    if p % 2 == 0: return None       # Page is absent
    p = (p >> 10) << 10              # clear low 10 bits (flags)
    # p is now the physical address of a page frame

    return p + o                     # physical frame + page offset
```

Listing 1: Virtual address translation algorithm. `>>` is (logical) right shift; `M.mem.read_u32(a)` reads a 32-bit value from the physical memory at address `a`.

2.3. Invariant of the Machine

Questions:

1. Specify the invariant of the machine.
2. Prove that, given your invariant, two **different** virtual addresses `v1` and `v2`, if `p1 = translate(M, v1)` and `p2 = translate(M, v2)` are both defined (i.e., not `None`), then `p1` $\neq$ `p2` are two different physical addresses.

3. Allocation (Q2)

Listing 2 and 3 define the page allocation algorithms as well as helpers:

- `zero_page_out` is a helper that receives a physical address of a page and writes zeros to the entire page;
- `alloc_meta_page` is a helper that allocates a new page in the metadata region by bumping `m_next`, and returns the physical address of the allocated page;
- `alloc_heap_page` is a similar helper for the heap region, bumping `h_next`; and
- `alloc` is the main allocation function that allocates a new heap page for the user, and updates page tables accordingly to map it to a new virtual address.

Question: specify and verify `alloc`, proving that, unless the page tables are full:

- it preserves the invariant of the machine, and
- the virtual address returned by `alloc` is mapped to a fresh physical page that was not previously allocated.

pseudocode

```
# Helper function to zero out a page of memory, given its physical address.
def zero_page_out(M: Machine, paddr: u32):
  assert paddr % 1024 == 0, "Physical address must be page-aligned"
  for i in range(256):
      M.mem.write_u32(paddr + i*4, 0)

# Helper function to create a fresh page table in the metadata region,
# and return its physical address.
def alloc_meta_page(M: Machine) -> u32:
  assert M.m_next < h_start, "Heap page allocation failed: out of memory"
  paddr = M.m_next
  M.m_next += 1024
  zero_page_out(M, paddr)
  return paddr

# Helper function to create a fresh page table in the heap region,
# and return its physical address.
def alloc_heap_page(M: Machine) -> u32:
  # A bit stricter check to prevent overflow of the bump pointer.
  assert M.h_next < (2**32 - 1024), "Heap page allocation failed: out of memory"
  paddr = M.h_next
  M.h_next += 1024
  zero_page_out(M, paddr)
  return paddr
```

Listing 2: Helper functions for allocation.

```pseudocode
# Allocates a new page for the user, updating page tables accordingly,
# and returns the virtual address of the allocated page.
def alloc(M: Machine) -> u32:
  # 1. Allocate a new heap page.
  new_page_paddr = alloc_heap_page(M)

  # 2. Update page tables to map a new virtual address to the allocated page.
  # This is done by walking through the page tables to find the first free slot,
  # and allocating new page tables in the metadata region as needed.
  for k in range(64):
    t = M.mem.read_u32(k*4)                  # read top-level directory entry
    if t % 2 == 0:                           # mid-level directory absent:
      mid_paddr = alloc_meta_page(M)           # allocate a new mid-level directory page
      M.mem.write_u32(k*4, mid_paddr | 1)      # mark as Present and write to directory
    else:
      mid_paddr = (t >> 10) << 10              # otherwise, clear flags to get address

    for i in range(256):                     # Same process for mid-level directory
      m = M.mem.read_u32(mid_paddr + i*4)
      if m % 2 == 0:
        leaf_paddr = alloc_meta_page(M)
        M.mem.write_u32(mid_paddr + i*4, leaf_paddr | 1)
      else:
        leaf_paddr = (m >> 10) << 10

      for j in range(256):                   # Same process for leaf page tables
        p = M.mem.read_u32(leaf_paddr + j*4)
        if p % 2 == 0:                       # free slot found: install the new page
          M.mem.write_u32(leaf_paddr + j*4, new_page_paddr | 1)
          return (k << 26) | (i << 18) | (j << 10)  # reconstruct vaddr

  assert False, "Page table full: no free virtual address"
```

Listing 3: User allocation algorithm: allocates a new heap page, and updates page tables to map it to a new virtual address.

4. A Machine with Multiple Processes (Q3)

The machine described so far assumes a single process is running on the machine. In practice, an MMU supports multiple processes and ensures that they are isolated from each other. This is typically done by maintaining a separate top-level page directory for each process, and switching the active top-level page directory when switching processes.

In our model, since the top-level page directory is indexed only by 6-bits (see Table 1), a process can only use 64 entries in this directory, representing 256 bytes. This means we can fit three other top-level page directories in the first page of memory, allowing for a total of 4 processes. The resulting memory layout is shown in Figure 4.

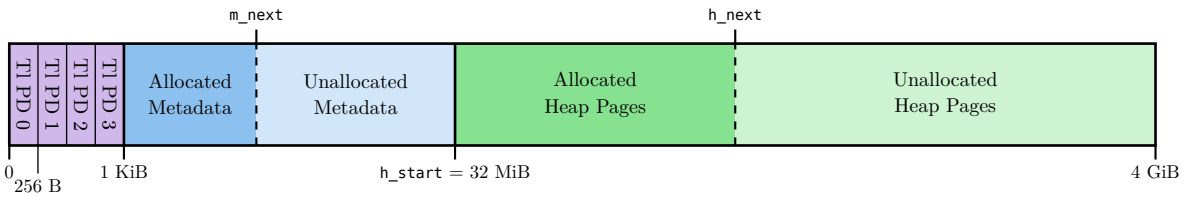

Figure 4: Physical memory layout for a machine with several processes (not to scale).

All four processes share the same physical memory, and the same allocator state (i.e., the same `m_next` and `h_next`). However, because each process has its own top-level page directory, the mid-level and leaf page tables, as well as the heap pages are all disjoint between processes, ensuring memory isolation.
Translating a virtual address for a process `p` $\in [0, 3]$ is done by using the `translate` function as before, but with the top-level page directory located at address `p * 256` instead of `0`. Similarly, the `alloc` function can be adapted by modifying the first loop to iterate over the 64 entries of the correct process. The full algorithms, with modifications for the multi-process setting highlighted, are given in Listing 4 and Listing 5.

Question:

1. Specify the invariant of the machine in the multi-process setting.
2. Prove that this invariant preserves the property previously proven for the single-process setting that two different virtual addresses are mapped to two different physical addresses (for the same process).
3. Prove that, given your invariant, given two virtual addresses `v1` and `v2` (that may be equal), and two different processes `proc_1` and `proc_2`, if `p1 = translate(M, proc_1, v1)` and `p2 = translate(M, proc_2, v2)` are both defined (i.e., not `None`), then `p1` $\neq$ `p2` are two different physical addresses.
4. Prove that the modified `alloc` function preserves the invariant of the machine, and that the virtual address returned by `alloc` is mapped to a fresh physical page that was not previously allocated, giving exclusive ownership of the allocated page to the process that called `alloc`.

Notes:

- *While we consider several processes, this question still assumes all operations to be running sequentially (i.e., no concurrency).*
- *The helper functions of Listing 2 are process-agnostic and reused without modification.*

```pseudocode
# Returns: physical address, or None if any page table entry is absent.
def translate(M: Machine, proc: u32, vaddr: u32) -> u32 | None:
    k = vaddr >> 26                  # Offset within top-level directory (6 bits)
    i = (vaddr >> 18) % 256          # Offset within mid-level directory (8 bits)
    j = (vaddr >> 10)  % 256         # Offset within page table (8 bits)
    o = vaddr % 1024                 # Offset within page (10 bits)

    # The top-level directory for process p is located at physical address proc*256.
    t = M.mem.read_u32(proc*256 + k*4) # "* 4" because each entry is 4 bytes
    if t % 2 == 0: return None       # Page is absent
    t = (t >> 10) << 10              # clear low 10 bits (flags)
    # t is now the physical address of a mid-level directory

    m = M.mem.read_u32(t + i*4)      # read mid-level directory entry
    if m % 2 == 0: return None       # Page is absent
    m = (m >> 10) << 10              # clear low 10 bits (flags)
    # m is now the physical address of a page table

    p = M.mem.read_u32(m + j*4)      # read page table entry
    if p % 2 == 0: return None       # Page is absent
    p = (p >> 10) << 10              # clear low 10 bits (flags)
    # p is now the physical address of a page frame

    return p + o                     # physical frame + page offset
```

Listing 4: Virtual address translation algorithm for multiple processes. Highlighted lines show modifications from the single-process version (Listing 1). `p` $\in [0, 3]$ is the process identifier.

```pseudocode
# Allocates a new page for process p, updating page tables accordingly,
# and returns the virtual address of the allocated page.
def alloc(M: Machine, p: u32) -> u32:
  assert 0 <= p and p <= 3, "Process identifier must be in [0, 3]"
  # 1. Allocate a new heap page.
  new_page_paddr = alloc_heap_page(M)

  # 2. Update page tables to map a new virtual address to the allocated page.
  # The top-level directory for process p starts at address p * 256.
  for k in range(64):
    t = M.mem.read_u32(p * 256 + k*4)       # read top-level directory entry
    if t % 2 == 0:                          # mid-level directory absent:
      mid_paddr = alloc_meta_page(M)          # allocate a new mid-level directory page
      M.mem.write_u32(p * 256 + k*4, mid_paddr | 1)  # mark Present and write
    else:
      mid_paddr = (t >> 10) << 10             # otherwise, clear flags to get address

    for i in range(256):                    # Same process for mid-level directory
      m = M.mem.read_u32(mid_paddr + i*4)
      if m % 2 == 0:
        leaf_paddr = alloc_meta_page(M)
        M.mem.write_u32(mid_paddr + i*4, leaf_paddr | 1)
      else:
        leaf_paddr = (m >> 10) << 10

      for j in range(256):                  # Same process for leaf page tables
        p_entry = M.mem.read_u32(leaf_paddr + j*4)
        if p_entry % 2 == 0:                # free slot found: install the new page
          M.mem.write_u32(leaf_paddr + j*4, new_page_paddr | 1)
          return (k << 26) | (i << 18) | (j << 10)  # reconstruct vaddr

  assert False, "Page table full: no free virtual address"
```

Listing 5: User allocation algorithm for multiple processes. Highlighted lines show modifications from the single-process version (Listing 3). $\texttt{p} \in [0, 3]$ is the process identifier.

5. A Pinch of Concurrency (Q4)

So far, we have assumed that all operations were running sequentially. In this question, we add a pinch of concurrency by allowing all processes to run concurrently, and interleave their operations arbitrarily. To ensure memory safety in this setting, we simply need to add two locks `m_lock` and `h_lock` to the machine state, and acquire these locks in the helper functions `alloc_meta_page` and `alloc_heap_page`, as shown in Listing 6 (with modifications highlighted).

Question: show the correctness of the modified `alloc` function when all processes can run concurrently.

Note: We assume that each process runs a single thread, a single process cannot allocate twice in parallel.

```
struct Machine {                                                    pseudocode
  mem: byte[2**32],
  m_next: u32,
  m_lock: Lock,
  h_next: u32,
  h_lock: Lock,
}

# Helper function to create a fresh page table in the metadata region,
# and return its physical address.
def alloc_meta_page(M: Machine) -> u32:
  M.m_lock.acquire()  # acquire lock to ensure exclusive access to m_next
  assert M.m_next < h_start, "Heap page allocation failed: out of memory"
  paddr = M.m_next
  M.m_next += 1024
  M.m_lock.release()  # release lock after updating m_next
  zero_page_out(M, paddr)
  return paddr

# Helper function to create a fresh page table in the heap region,
# and return its physical address.
def alloc_heap_page(M: Machine) -> u32:
  M.h_lock.acquire()  # acquire lock to ensure exclusive access to h_next
  # A bit stricter check to prevent overflow of the bump pointer.
  assert M.h_next < (pow(2, 32) - 1024), "Heap page allocation failed: out of memory"
  paddr = M.h_next
  M.h_next += 1024
  M.h_lock.release()  # release lock after updating h_next
  zero_page_out(M, paddr)
  return paddr
```

Listing 6: Helper functions for allocation in the concurrent setting.

6. Recursive page tables

For Challenge 4, you may either continue on your solution to Challenge 3 or start solving this part of the problem, which you should be able to do somewhat independently of the previous sections.

6.1. Motivation

In real computers, the mapping from virtual to physical addresses is handled directly by the hardware. After a bit of setup, the processor "switches" to virtual memory; this is done once. After switching, **all** memory accesses are translated. In other words, the functions `M.mem.read_u32` and `M.mem.write_u32` cannot be used anymore; instead, algorithms must use `M.readv_u32` and `M.writev_u32` which read and write to virtual addresses, and follow the following equivalences:

```pseudocode
M.readv_u32(v) <=> M.mem.read_u32(translate(v))
M.writev_u32(v, x) <=> M.mem.write_u32(translate(v), x)
```

In turn, this means that functions that manipulate the memory, such as `alloc`, now need to use virtual addresses. This discipline also spans over metadata pages: one needs the virtual address of a page table to update it when allocating!
One elegant solution to this problem is to use *recursive page tables* where the *first* entry of each page table (or page directory) points to itself.

6.2. Updated algorithms

Before switching to virtual memory mode, a one-time initialization sets the self-referential entry:

```pseudocode
# One-time setup (called once, still in physical addressing mode).
def preinit(M: Machine):
    M.mem.write_u32(0, 0 | 1)   # PD[0] = physical address 0, Present=1
```

Listing 7: Initialization: entry 0 of the top-level page directory is set to point to the directory itself. After this call, the processor switches to virtual addressing mode and all subsequent accesses use `readv_u32` / `writev_u32`.

Because `PD[0]` always points back to the top-level page directory, address translation of a virtual address with $K = 0$ starts by following the self-reference. This makes it possible to reach *any* level of the page-table hierarchy using a purely virtual address:

- To read or write **top-level PD entry** k: translation uses $K = 0, I = 0, J = 0$ and page offset $O = k \cdot 4$, i.e. virtual address $k \cdot 4$.
- To read or write **mid-level PD entry** i under top-level entry k: use $K = 0, I = 0, J = k$, $O = i \cdot 4$, i.e. virtual address $(k \ll 10) \mid (i \cdot 4)$.
- To read or write **leaf PT entry** j under top-level k and mid-level i: use $K = 0, I = k, J = i$, $O = j \cdot 4$, i.e. virtual address $(k \ll 18) \mid (i \ll 10) \mid (j \cdot 4)$.

For convenience we define three helper functions that encapsulate the recursive virtual addresses:

```pseudocode
# Return the virtual address of top-level PD entry k.
def va_pd(k: u32) -> u32:
    return k * 4

# Return the virtual address of mid-level PD entry i under top-level k.
def va_mid(k: u32, i: u32) -> u32:
    return (k << 10) | (i * 4)

# Return the virtual address of leaf PT entry j under top-level k, mid i.
def va_leaf(k: u32, i: u32, j: u32) -> u32:
    return (k << 18) | (i << 10) | (j * 4)
```

Listing 8: Helper functions for computing recursive virtual addresses of page-table entries.

Listing 9 and 10 give the updated helper functions and `alloc`. The key changes are:

- `zero_page_out` is replaced by `vzero_page_out`, which zeroes all 256 entries of a page via virtual writes.
- `alloc_meta_page` and `alloc_heap_page` now take no extra arguments: each simply bumps the appropriate pointer and returns the physical address of the new page, mirroring Listing 2 exactly.
- `alloc` handles all virtual writes: after allocating a metadata or heap page it installs the physical address into the parent entry, then immediately calls `vzero_page_out` using the recursive virtual address — which is only valid after the install. All memory accesses are virtual; there are no physical writes.

6.3. Questions

Note: we assume a single process without concurrency.

1. Specify the new invariant of the machine in the recursive page table setting.
2. Prove that the new invariant preserves the injectivity property of address translation: two different virtual addresses are mapped to two different physical addresses (for the same process).
3. Prove that the modified `alloc` function preserves the invariant of the machine, and that the virtual address returned by `alloc` is mapped to a fresh physical page that was not previously allocated, giving exclusive ownership of the allocated page to the process that called `alloc`.

pseudocode

```
# Zeros all 256 entries of the page at the given virtual address.
def vzero_page_out(M: Machine, vaddr: u32):
  assert vaddr % 1024 == 0, "Virtual address must be page-aligned"
  for i in range(256):
      M.writev_u32(vaddr + i*4, 0)

# Allocates a fresh metadata page and returns its physical address.
def alloc_meta_page(M: Machine) -> u32:
  assert M.m_next < h_start, "Metadata space exhausted"
  paddr = M.m_next
  M.m_next += 1024
  return paddr

# Allocates a fresh heap page and returns its physical address.
def alloc_heap_page(M: Machine) -> u32:
  assert M.h_next < (2**32 - 1024), "Heap space exhausted"
  paddr = M.h_next
  M.h_next += 1024
  return paddr
```

Listing 9: Updated helper functions for the recursive page table setting. `alloc_meta_page` and `alloc_heap_page` are unchanged in structure from Listing 2. `vzero_page_out` uses virtual writes. All installs and zeroing of newly allocated pages are done in `alloc` (Listing 10).

```pseudocode
# Allocates a new page for the user, updating page tables accordingly,
# and returns the virtual address of the allocated page.
def alloc(M: Machine) -> u32:
  for k in range(64):
    if M.readv_u32(va_pd(k)) % 2 == 0:        # mid-level directory absent
      mid_paddr = alloc_meta_page(M)
      M.writev_u32(va_pd(k), mid_paddr | 1)   # install in top-level PD (virtual)
      vzero_page_out(M, va_mid(k, 0))             # zero mid-level page (now mapped)
    else:
      mid_paddr = (M.readv_u32(va_pd(k)) >> 10) << 10

    for i in range(256):
      if M.readv_u32(va_mid(k, i)) % 2 == 0:  # leaf page table absent
        leaf_paddr = alloc_meta_page(M)
        M.writev_u32(va_mid(k, i), leaf_paddr | 1)  # install in mid-level PD (virtual)
        vzero_page_out(M, va_leaf(k, i, 0))         # zero leaf page (now mapped)

      for j in range(256):
        if M.readv_u32(va_leaf(k, i, j)) % 2 == 0:  # free slot
          heap_paddr = alloc_heap_page(M)
          va_page = (k << 26) | (i << 18) | (j << 10)
          M.writev_u32(va_leaf(k, i, j), heap_paddr | 1)  # install (virtual)
          vzero_page_out(M, va_page)                       # zero heap page (now mapped)
          return va_page

  assert False, "Page table full: no free virtual address"
```

Listing 10: Updated allocation algorithm for the recursive page table setting. Highlighted lines show modifications from Listing 3. After installing each new metadata page, `vzero_page_out` is called immediately using the recursive virtual address — only valid once the page is mapped. All memory accesses are virtual; there are no physical writes.