CS-202 Exercises on Memory (L04 - L05)

Exercise 1: Processes don't share virtual address space

Modern systems run multiple processes simultaneously instead of one at a time. However, one process should not be able to modify another process's memory unless the two processes agree. We will see how this protection is enabled through separate virtual address spaces for each process.

Suppose process *P1* is executing the following program, which forks a child process *P2*.

```
int x = 0;                    // A global variable
int main() {
   x = 10;
   pid_t pid = fork();
   if (pid == 0) {            // Child process P2
      x = 42;
      printf("Child P2 says x = %d\n", x);
   } else {                   // Parent process P1
      wait(pid); // Wait for the child P2 to finish
      printf("Parent P1 says x = %d\n", x);
  }
  return 0;
}
```

- If `x` is at virtual address VA_x in process P1, what is its corresponding address in process P2? The virtual address of x in process P2 is the same as in process P1: VA_x.
- If VA_x maps to physical address PA_x in process P1, what does it map to in process P2? Before modification, it is PA_x as well, after modification, it maps to a new PA_x' (essentially copy on write).
- What does P1 print and what does P2 print? P1 print : Parent P1 says x = 10; P2 print: Child P2 says x = 42
- Under the simplifying assumption that there is no CPU cache, what must the CPU do in order to write value to the memory location that stores variable `x`?

  The CPU must perform the following operations:
  - Translate the virtual address (VA_x) to a physical address (PA_x) using the page table (specifically, MMU inside the CPU does this).
  - Check access permissions (e.g., is the page writable?).
  - Issue a memory write operation to store the new value at PA_x.
  - If copy-on-write (COW) applies, allocate a new page (PA_x'), copy the old contents, update the page table for the process, and write to the new page.

- Suppose process P1 is running, then the timer interrupt occurs and the OS scheduler picks P2 to run.
- What information must be updated concerning memory accesses?
  - Update the Page Table Base Register (i.e., register cr3) to point to P2's page table. (The address of this page table is stored in a per process task struct in linux for P2).
  - Update CPU registers to restore P2's execution state, including its stack pointer, instruction pointer, and general-purpose registers.

- [Optional, for the students who took computer architecture] Removing the simplifying assumption that there is no CPU cache; what information needs to be updated such that caching works correctly with paging? What if the system uses segmentation for address translation?

  For paging:
  - The TLB (Translation Lookaside Buffer) must be flushed or updated when switching processes to prevent using invalid address translations.
  - If TLB has ASID (address space id), such flush is not necessary.

  For segmentation:
  - The segment descriptor tables must be switched to match the new process's memory layout.

Exercise 2: Threads do share virtual address space

Besides creating a new process with fork, we can create a new thread within the same process with pthread_create and wait for a thread to finish with pthread_join. Given a slightly different program below, answer the following questions:

```
int x = 0;
// The "child" thread function
void* child_thread(void* arg) {
     x = 42;
     printf("Child thread says x = %d\n", x);
     return NULL;
}

int main() {
     . . .
     x = 10;
    // Create child thread pthread_t tid;
    pthread_create(&tid, NULL, child_thread,
NULL);
    // Wait for child thread to finish
    pthread_join(tid, NULL);

    printf("Parent thread says x = %d\n", x);
    return 0;
}
```

- If x is at virtual address VA_x in the parent thread, what is its corresponding address in the child thread? The virtual address of x in the child thread is the same as in the parent thread: VA_x.
- If VA_x maps to physical address PA_x in the parent thread, what does it map to in the child thread? Since both threads share the same memory space, VA_x in the child thread also maps to PA_x.
- What does the parent thread print, and what does the child thread print?
    - Child thread prints: Child thread says x = 42
    - Parent thread prints: Parent thread says x = 42

Exercise 3: Address translation using base & bound

Consider a system that implements address translation using the base & bound approach. Consider the following code:

```
char *a = 0x108;

Putchar (*(a + 0x10));

putchar(*0x128);
```

For the two lines with putchar, describe which virtual address will be referenced and which physical address will be accessed, if the corresponding process:

(a) Has base 0x100 and bound 0x230.

`putchar(*(a + 0x10))`: the virtual address a + 0x10 is 0x108+0x10 = 0x118, which is within the range. The physical address is 0x118+0x100 = 0x218

`putchar(*(0x128))`: the virtual address is 0x128, which is within the range. The physical address is 0x128+0x100 = 0x228

(b) Has base 0x250 and bound 0x370

`putchar(*(a + 0x10))`: the virtual address a + 0x10 is 0x108+0x10 = 0x118, The physical address is 0x118+0x250 = 0x368.which is within the range. `putchar(*(0x128))`: the physical address is outside the range, so a segmentation fault will occur.

Exercise 4: Address translation using a single-level (linear) page table

Consider a system that implements address translation using paging and, in particular, a single-level (linear) page table. Assume page-table entries of 4 bytes.

Assume 32-bit virtual and physical addresses, and pages of 4 KiB (4096 bytes). In each page-table entry (PTE), how many extra bits are available for storing additional information other than the frame number?

Step1: since the page size is 2^12 bytes = 4KiB, the number of bits for the offset within the page is 12 bits. In a 32-bit address space, the number of bits for the page number (and also the frame number in the physical address space) is 32−12=20 bits.
Step2: each page-table entry (PTE) is 4 bytes (32 bits). Since the PPN takes 20 bits, the remaining bits for extra information are: 32 − 20 = 12 extra bits.
So, 12 extra bits are available for storing additional information

Assume 40-bit virtual and physical addresses, and pages of 4 KiB. Does your answer change and – if yes – how?

Page size is still 4 KiB, thus 12 bits for the page offset. The virtual address space is 40 bits, so the remaining 40 - 12 = 28 bits are used for indexing the page table. So the PPN takes 40 - 12 = 28 bits.
Each PTE is still 4 bytes (32 bits). Since the PPN takes 28 bits,
So the remaining bits for extra information are 32 − 28 = 4 extra bits.

Assume 8-bit virtual addresses, 11-bit physical addresses, and pages of 32 bytes.

- How many bits are used for computing the page offset? How many bits are left for indexing the page table (computing the virtual page number)?

  Since 32 = 2^5, we need 5 bits for the offset within the page. There are 8-5=3 bits left to index into the page table.

- How many entries does each page table have in total?

  The number of entries in the page table is 2^3=8 entries

Consider the last set of assumptions (8-bit virtual addresses etc). Assume additionally that the first bit of each PTE is a “valid” bit, which by default is set to 0 (invalid).

Suppose a process performs the following two operations:

- It stores a one-byte value at virtual address `0x82`, and this value ends up in physical frame number `0b1100.`
- It stores a 64-byte array starting at virtual address `0xA0`, and this array ends up stored in physical frame numbers `0b11000` & `0b11`.

Fill out the following page table after these two operations have occurred.

Valid Bit	Virtual Page Number (VPN)	Page Frame Number (PFN)
0	0	-
0	1	-
0	2	-
0	3	-
1	4	12
1	5	24
1	6	3
0	7	-

Exercise 5: Address translation using a multi-level page table

Consider a system with 16-bit virtual addresses, pages of 256 bytes, and a two-level page table.

- What is the total amount of physical memory that could be referenced by such a page table? 16(L1 entries)×16(L2 entries)×256(bytes per page)=65536 bytes=64 KB
- [Advanced] Suppose the given system has X bytes of free physical memory, where X is the answer you gave to the first question. Assume no swapping to disk.

  Suppose the user invokes a program, which triggers the creation of a new process. Can this new process use all X available bytes to store its code and data? (Hint: what is one of the disadvantages of using paging?)

  No, because the OS also needs some bytes of memory to store page tables of this process in order for the virtual memory translation to work.

  A more detailed calculation is the following:

  First, there needs one first level page table, that stores addresses to second level page tables, and it occupies 16*8 = 128 bytes. Note, in a real OS, one page table normally occupies 1 page. Here, one page table occupies 128 bytes, which are less than one page.

  Now, let's assume that all entries of this first level page table are not empty and point to a second level page table, then there are 16 second level page tables. In total, 17 page tables which occupies 17*128 bytes = 2176 bytes. In total, these page tables stored translation for 65536 bytes of memory.

  However, the process can't use all 65536 bytes of memory, because then no memory can be used to store the page tables.

  Hence, some entries of the first level page table should be empty.

  Now let's assume the process can use Y pages. Each 16 pages the process uses requires one second level page table. Hence the following inequality should hold:

  Y*256 + Y/16*128 + 128 < 65536 implies

  Y <= 247 pages implies

  the process can use 247*256 bytes of memory.

  Using the partial page tables shown below, translate virtual address `0x1f3a` into a physical address.

  - 0x1F3a=0b0001_1111_0011_1010, page size is 8-bit (4-bit for first level page table and another 4-bit for second-level page table

- the first-level page table index is 0b0001 = 0x1
- the second-level page table index is 0b1111 = 0xF
- The 8-bit page offset is the right-most 8 bits, which is 0x3a
- 0x1 → 0x6000 → level 2: 0xF → 0xff00 + 0x3a = 0xff3a

The level-1 page table:

Index	Address
0x0	0x2000
0x1	0x6000
…	…
0xf	0x1300

The level-2 page table at physical address `0x6000` is:

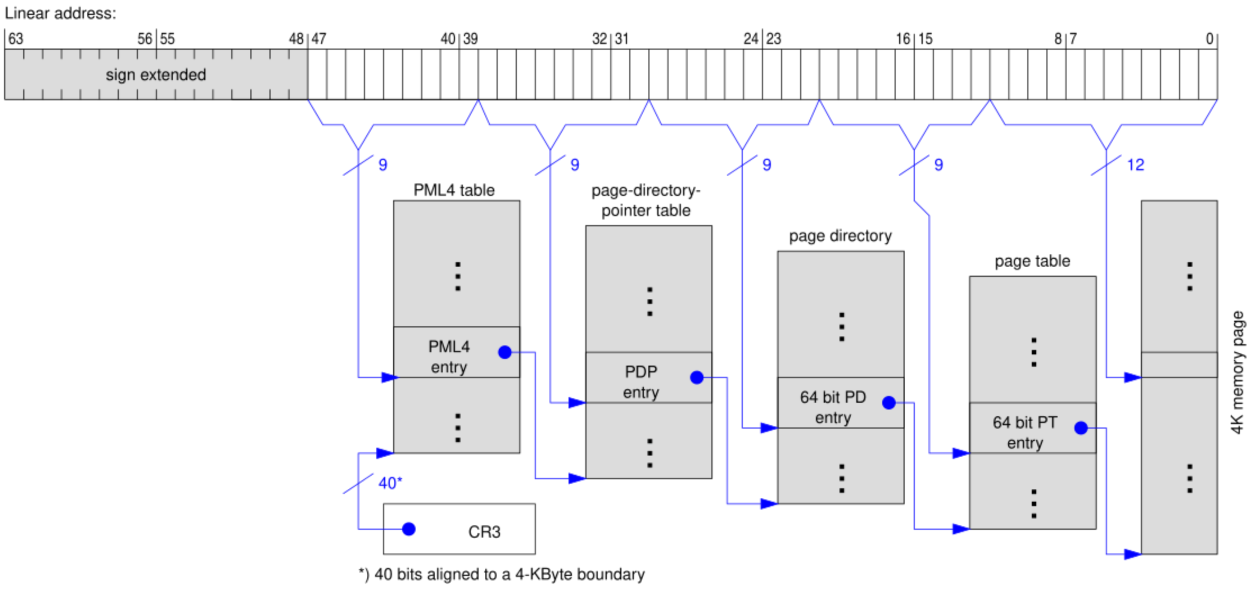
Index	Address
0x0	0x3000
0x1	0x7000
…	…
0xf	0xff00

Exercise 6: What are possible page sizes?

The following diagram illustrates which bits of each virtual address are used to index each page-table level, in a given computer architecture.

Given this architecture, one cannot change which virtual-address bits are used to index each page-table level. E.g., to index the first page-table level, bits 39 to 47 must be used; to index the second page-table level, bits 30 to 38 must be used; and so on. However, the OS is free to write anything it wants inside each page table.

- What page sizes other than 4KiB can be used (without reducing the amount of physical memory that can be referenced)?
    1. Page size: 4KiB, offset bits: 12 bits, page table levels used: PML4-PDP-PD-PT
    2. Page size: 2MiB, offset bits: 21 bits, page table levels used: PML4-PDP-PD
    3. Page size: 1GiB, offset bits: 30 bits, page table levels used: PML4-PDP
- What are the benefits of using larger pages instead of smaller ones? (Consider trade-offs in TLB (Translation Lookaside Buffer) efficiency, memory fragmentation, and access latency.)
  Minimizes TLB overhead significantly, which reduces TLB miss and hence improves latency.
  Best for large-memory workloads like high-performance computing (HPC), machine learning, and virtualization.



Exercise 7: Digesting the benefits of virtual memory

Consider the following scenario:

- A system with 16-bit virtual addresses.
- Only 32 KiB of physical memory have been installed.
- Three processes, P1, P2, and P3, are started, each with a memory image of 20 KiB.
- During its lifetime, each process only accesses disjoint 4 KiB sections of its 20 KiB memory image.

- Do you think it would be possible to host all three processes simultaneously without memory virtualization? If yes, how? If not, why not?

  In pure physical addressing (without virtual memory): Each process must be assigned a fixed region in physical memory. The OS must fit all allocated memory within 32 KB RAM. Since the total request is 60 KB, but RAM is only 32 KB, it is impossible to run all three simultaneously.

- With memory virtualization, is it possible to host all three processes simultaneously? If yes, how? If not, why not?

  Each process sees its own 64 KB address space (even if only 32 KB of physical memory exists). The OS swaps or maps only the required 4 KB working set into physical RAM. At any moment, only the needed 4 KB per process is kept in RAM, while the rest remains on disk or is swapped out. Total active memory usage: 4 KB(A)+4 KB(B)+4 KB(C)=12 KB. Since 12 KB < 32 KB physical RAM, all three can run simultaneously.

Exercise 8: Base & bound against paging

(a) Basic questions:

- What is the main advantage of implementing address translation using paging as opposed to base & bound?

  The primary advantage of paging over base & bound is efficient memory utilization and fragmentation management:

  - Relieve fragmentation: Paging divides memory into fixed-sized pages, allowing processes to use non-contiguous physical memory, which reduces wasted space compared to base & bound, which requires a single contiguous block.
  - Paging enables demand paging, where only needed pages are loaded into RAM, allowing larger programs to run even if physical memory is limited.
  - Paging makes it easier to support multiple processes by dynamically allocating memory without requiring contiguous allocation.
  - …

- What is the main advantage of using base & bound as opposed to paging?

  The main advantage of base & bound is lower overhead and faster address translation:

  - Lower hardware complexity: Base & bound only requires a base register and a bound register, making address translation significantly faster compared to paging, which involves multiple memory accesses due to page table lookups.

  - Less memory overhead: Paging requires a page table, which consumes additional memory, whereas base & bound only needs two registers.
  - Faster context switching: Since base & bound uses just two registers, switching between processes requires only updating these registers, making it faster compared to paging, where switching might need to flush the TLB.

- Think of a scenario where base & bound might be preferable to paging. Explain your reasoning.

  A scenario where base & bound is preferable is in real-time embedded systems that require ultra-low latency and deterministic performance:

  - Real-time constraints: Systems such as avionics, industrial automation, and medical devices require predictable and fast memory access. Paging introduces variable delays due to page table lookups and possible page faults, which could be unacceptable in time-critical applications.
  - Minimal hardware resources: Many embedded systems have limited memory and cannot afford the overhead of maintaining page tables. Base & bound, requiring only two registers, is more efficient in such constrained environments.
  - Single-task environments: If the system runs a single dedicated task or a few well-defined processes with preallocated memory, the benefits of paging (e.g., memory sharing and dynamic allocation) are unnecessary, making base & bound a simpler and more efficient choice.
  - …

(b) Design: Imagine you are tasked with designing a new system that can concurrently run multiple operating systems. Your design must handle memory management efficiently while ensuring strong security and high performance. You can choose between base & bound and paging for virtual memory management.

Overall reasoning: Because each guest OS already uses paging internally to manage its processes, it does not matter whether the underlying system provides a single large contiguous region or subdivided page frames. By simply giving each guest OS a big contiguous memory segment, the new system's job is simplified (it only needs to handle base & bound registers). Therefore, base & bound is preferred here for its simplicity. Each guest OS can still do fine-grained paging internally if needed.

- How does your chosen approach optimize memory access and overall system speed? (Consider the overhead involved in address translation and the potential impact on cache performance.)

  Since the new system uses base & bound, switching to a new virtual address space (e.g., when switching between guest OSes) is very fast and low-cost:

  The address translation simply adds the base to the virtual address if it is within the bound—far fewer steps than walking multi-level page tables.

There is no need to load or flush Translation Lookaside Buffers (TLBs) at this new system level, reducing potential overhead and avoiding TLB contention.

- Discuss how your design tradeoff simplifies memory allocation/deallocation and minimizes fragmentation.

  With base & bound, memory is allocated in contiguous chunks, one chunk per OS. This means:
  - The new system only has to manage continuous blocks of physical memory.
  - Each guest OS is then responsible for handling its own internal paging and any fragmentation within its assigned chunk. Hence fragmentation is less of a concern for the new system.
  - Allocation and deallocation are quick.
- Explain how your chosen method provides effective isolation between the operating systems

  Different OSes live in disjoint physical memory, with access controlled by base & bound. This provides effective isolation.

[Advanced] Exercise 9: Where does the virtual address of a program come from?

Have you ever wondered where the virtual addresses of your program come from? In class, we said that the compiler compiles a source code into an executable program. To be more precise, the compiler compiles source code into an intermediate form, which is almost an executable program, but not quite; there is another tool, called the "linker", which prepares this intermediate form into an executable program. The linker plays a crucial role in determining the virtual memory layout of the program's code and data: it uses a "linker script" to define the memory locations of the different segments of the memory image,, such as `.text`, `.rodata`, `.data`, and `.bss`

```
OUTPUT_FORMAT("elf64-x86-64")
ENTRY(_start)

SECTIONS
{
  . = 0x400000;              /* Default load address for ELF */
  .text : { *(.text) }      /* Code section */
  .rodata : { *(.rodata) } /* Read-only data section */
  .data : { *(.data) }       /* Initialized data section */
  .bss : { *(.bss) }         /* Uninitialized data section */
  . = ALIGN(16);              /* Align to 16 bytes */
  _end = .;                   /* Symbol marking the end of the program */
}
```

In this script:

- The `.text` section (code) starts at the virtual address `0x400000`.
- The `.rodata` section (read-only data) is placed immediately after the `.text` section.
- The `.data` section (initialized data) is placed after `.rodata`.
- The `.bss` section (uninitialized data) is placed after `.data`.
- The linker aligns the sections to 16 bytes and marks the end of the program with the symbol _end

- If the `.text` section occupies `0x1000` bytes, the `.rodata` section occupies `0x500` bytes, the `.data` section occupies `0x200` bytes, and the `.bss` section occupies `0x300` bytes, what are the starting virtual addresses of each section? find the starting virtual addresses of each section and the value of `_end`.

  1. .text Section: Start address + size = 0x400000 + 0x1000 = 0x401000.

  2. .rodata Section: Start address + size = 0x401000 + 0x500 = 0x401500. Align to 16 bytes → Round up 0x401500 to the next multiple of 0x10: 0x401500 (already aligned).

  3. .data Section: Start address + size = 0x401500 + 0x200 = 0x401700. Align to 16 bytes → 0x401700 is already aligned.

  4. .bss Section: Start address + size = 0x401700 + 0x300 = 0x401A00. Align to 16 bytes → 0x401A00 is already aligned.

  5. The address after .bss is 0x401A00. _end is then assigned the value of the next available address, which is:_end=0x401A00`

- Modify the script to add a new section `.custom` (aligned, occupies `0x100` bytes) between `.data` and `.bss`. Find the new value of `_end`.

  .custom section : start address + size = 0x401700 + 0x100 = 0x401800. 0x401800 is already aligned.

  .bss Section: start address + size = 0x401800 + 0x300 = 0x401B00, is already aligned.

  The .bss section ends at 0x401B00, so _end is assigned this value.

- Discuss implications if virtual address `0x400000` is already in use by another process.

  This is irrelevant, because each process has its own virtual address space.
