Sergio Garcia Tapia
Computer Systems: A Programmer's Perspective, by Bryant and O'Hallaron
Chapter 9: Virtual Memory
May 23, 2024

Practice Problems

Exercise 9.1. Complete the following table, filling in the missing entries, and replacing each question mark with the appropriate integer. Use the following units: K = 2^{10} (kilo), M = 2^{20} (mega), G = 2^{30} (giga), T = 2^{40} (tera), P = 2^{50} (peta), or E = 2^{60} (exa).

Number of VA bits (n)	Number of VA (N)	Largest possible VA
8	_____	_____
_____	$2^{?}$ = 64 K	_____
_____	_____	$2^{32} - 1$ = ?G - 1
_____	$2^{?}$ = 256 T	_____
64	_____	_____

Solution: A virtual address with $N = 2^n$ addresses is an n-bit address space. Note that

- $2^8 = \frac{1}{2^2}2^{10} = \frac{1}{4}$ K.
- 64 K = $2^6 \cdot 2^{10} = 2^{16}$
- If the largest address is $2^{32} - 1$, then $n = 32$. Thus $N = 2^{32} = 2^2 \cdot 2^{30}$, which is 4G.
- 256 T = $2^8 \cdot 2^{40}$, so $n = 48$. This is $\frac{1}{4}$ P.
- $2^{64} = 2^4 \cdot 2^{60} = 4$.

Number of VA bits (n)	Number of VA (N)	Largest possible VA
8	$2^8 = \frac{1}{4}$ K	$2^8 - 1 = \frac{1}{4}$ K $- 1$
16	2^{16} = 64 K	$2^{16} - 1$ = 64 K $- 1$
32	2^{32} = 4 G	$2^{32} - 1$ = 4 G - 1
48	2^{48} = 256 T	$2^{48} - 1$ = 256 T $- 1$
64	2^{64} = 4 E	$2^{64} - 1$ = 4 E $- 1$

Exercise 9.2. Determine the number of page table entries (PTEs) that are needed for the following combination of virtual address size (n) and page size (P):

n	$P = 2^p$	Number of PTEs
16	4K	_____
16	8K	_____
32	4K	_____
32	8K	_____

Solution: For $n = 16$, there are 2^{16} virtual addresses. According to Section 9.3, the conceptual arrangement of a virtual memory is as an array of N contiguous byte-size cells stored on disk. A page table size is given in bytes, so 4K means 2^{12} bytes. Since we use byte addressing, an address space with a size of 2^{16} can have $2^{16}/2^{12} = 2^4 = 16$ pages. If a page is 8K bytes in size, we can fit $2^{16}/2^{13} = 8$ pages instead.

If $n = 32$, then a 4K page size means $2^{32}/2^{12} = 2^{20} = 1$ M pages. If page size is 8K, then it's $2^{32}/2^{13} = 2^{19} = 2^9 \cdot 2^{10} = 512$K.

n	$P = 2^p$	Number of PTEs
16	4K	16
16	8K	8
32	4K	1M
32	8K	512K

Exercise 9.3. Given a 32-bit virtual address space and a 24-bit physical address, determine the number of bits in the VPN, VPO, PPN, and PPO for the following page sizes P.

Solution: The address pace uses $n = 32$ bits, so the size of the virtual address space is $N = 2^{32}$. The page is is $P = 2^p$ in bytes, where p bits are used for the VPO, and $n - p$ bits are used for the VPN. We also use the same p bits for the PPO. The number of addresses in the physical address space is given by $M = 2^m$. Then $m - p$ bits are used for the PPN.

In this problem, $n = 32$, and $m = 24$. When the page size is 1 KB, this means $P = 2^{10}$, meaning $p = 10$. Thus we use 10 VPO bits and $n - p = 22$ VPN bits. We also use $p = 10$ PPO bits, and we use $m - p = 14$ PPN bits.

For 2 KB pages, we have $p = 2^{11}$, so we use 11 VPO bits, 21 VPN bits, 10 PPO bits, and 13 PPN bits.

	Number of			
P	VPN bits	VPO bits	PPN bits	PPO bits
1 KB	22	10	14	10
2 KB	21	11	13	11
4 KB	20	12	12	12
8 KB	19	13	11	13

Exercise 9.4. Show how the example memory system in Section 9.6.4 translates a virtual address into a physical address and accesses the cache. For the given virtual address, indicate the TLB entry accessed, physical address, and cache byte value returned. Indicate whether the TLB misses, whether a page fault occurs, and whether a cache miss occurs. If there is a cache miss, enter "—" for "Cache byte returned". If there is a page fault, enter "—" for "PPN" and leave parts C and D blank.

Virtual address: `0x03d7`

Solution:

(a) Virtual address format: In the example, we used $n = 14$ bits for the virtual address space, and 1-byte words. The given address in hex can be translated to the following binary sequence:

`00 0011 1101 0111`

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	1	0	1	1	1

(b) Address translation: With virtual addresses that are $n = 14$ bits wide and page sizes that are $P = 64 = 2^6$ bytes in size, we use the lower 6 bits for the VPO and upper 8 bits for the VPN. Therefore, the VPN bits are `0000 1111`, which is hex `0x0f`, and the VPO bits are `01 0111`, meaning the VPO in hex is `0x17`. The lower 2 bits of the VPN are used for the TLBI, and the upper 6 bits are used for the TLBT. Thus, `11` is `0x03` for the TLBI, and `00 0011` is `0x03` also for the TLBT. It is a hit for the TLB, since it returns a PPN of `0x0d`. Thus there is no page fault.

Parameter	Value
VPN	`0x0f`
TLB index	`0x03`
TLB tag	`0x03`
TLB hit? (Y/N)	Yes
Page fault? (Y/N)	No
PPN	`0x0d`

(c) Physical Address Format:

11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	0	0	0	1	1	1

(d) Physical memory reference: Now we concatenate the 6-bit PPN `0x0D`, which is `00 1101` and the 6-bit VPO `0x17`, which is `01 0111`, to create the physical address `0011 0101 0111`, or `0x357`.

The MMU sends the physical address to the cache, which extracts the lowest 2 bits for the block offset (CO), the next 4 bits for the cache index (CI), and the highest 6 bits for the cache tag (CT). Thus, CO is `11` or `0x3`, the CI is `0101` or `0x5`, and the highest 6 bits are `00 1101`, or `0x0d`. The tag for the cache set with index `0x5` is `0x0d`, which does not which matches what we have, we have a cache hit, and the valid bit is set. Thus we use the offset, which is `0x3`, to get the byte in block 3, which is `0x1d`.

Parameter	Value
Byte offset	`0x3`
Cache index	`0x5`
Cache tag	`0x0d`
Cache hit ? (Y/N)	Yes
Cache byte returned	`0x1d`

Exercise 9.5. Write a C program `mmapcopy.c` that uses `mmap` to copy an arbitrary-size disk file to `stdout`. The name of the input file should be passed as a command-line argument.

Solution: See `./05-mmapcopy/mmapcopy.c`:

```c
#include <stdio.h> /* fprintf(), printf(), stderr */
#include <stdlib.h> /* exit() EXIT_FAILURE */
#include <string.h> /* strcmp(), strerror() */
#include <errno.h> /* errno */
#include <unistd.h> /* STDOUT_FILENO, write(), fstat() */
#include <sys/types.h> /* off_t */
#include <fcntl.h> /* open(), O_RDONLY */
#include <sys/mman.h> /* mmap() */
#include <sys/stat.h> /* struct stat, fstat() */

#define FILE_ARG_IDX 1

void
unix_error(char *msg)
{
    fprintf(stderr, "%s: %s\n", msg, strerror(errno));
    exit(EXIT_FAILURE);
}

int
main(int argc, char *argv[])
{
    /* Validate command-line arguments */
    if (argc != 2 || strcmp("--help", argv[1]) == 0) {
        fprintf(stderr, "Usage: %s filename\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    /* Open input file */
    int fd = open(argv[FILE_ARG_IDX], O_RDONLY);
    if (fd == -1)
        unix_error("Failed to open input file");

    /* Get file size */
    struct stat sb;
    if (fstat(fd, &sb) == -1)
        unix_error("Failed to get file size");

    /* Get memory map */
    off_t offset = 0; /* Start at beginning of the file */
    void *start = NULL; /* Let kernel decide */
    char *bufp = mmap(start, sb.st_size, PROT_READ, MAP_PRIVATE, fd, offset);
    if (bufp == MAP_FAILED)
```

```
        unix_error("Failed to create memory mapping for input sile");

    /* Write to stdout */
    if (write(STDOUT_FILENO, bufp, sb.st_size) == -1)
        unix_error("Failed to write contents to stdout");
    exit(EXIT_SUCCESS);
}
```

Exercise 9.6. Determine the block sizes and header values that would result from the following sequence of `malloc` requests. Assumptions: (1) The allocator maintains double-word alignment and uses an implicit free list with the block format from Figure 9.35. (2) Block sizes are rounded tot he nearest multiple of 8 bytes.

Solution: The format on Figure 9.35 requires a 4-byte header, where the 29 high-order bits encode the size, the least significant bit indicates whether or not it is allocated, and the next two bits are 0. Moreover we require a double-word alignment constraint, which means that block sizes must be a multiple of 8.

When 1 byte is requested by `malloc(1)`, we have a 4-byte header, 1 byte for the byload, 3 bytes of padding so that the block can satisfy the double-word constraint. Thus the header is `0x8 | 0x1 = 0x9`, where the lest significant 1 indicates that the block has been allocated.

When 5 bytes are requested, we again have a 4-byte header, 5 byte payload, and 7 bytes of padding to satisfy the double-word alignment constraint. Thus the block itself takes 16 bytes, meaning the header is `0x10 | 0x1 = 0x11`.

With `malloc(12)`, we have a 4-byte header, 12-byte payload, and there is no padding necessary because it already satisfies the alignment constraint. The header is `0x10 | 0x1 = 0x11`.

Lastly, with `malloc(13)`, we have a 4-byte header, 13-byte payload, and 7 bytes of padding to satisfy the double-word alignment constraint. The header is `0x18 | 0x1 = 0x19`.

Request	Block size (decimal bytes)	Block header (hex)
`malloc(1)`	8	`0x9`
`malloc(5)`	16	`0x11`
`malloc(12)`	16	`0x11`
`malloc(13)`	24	`0x19`

Exercise 9.7. Determine the minimum block size for each of the following combinations of alignment requirements and block formats. Assumptions: Implicit free list, zero-size payloads are not allowed, and headers and footers are stored in 4-byte words.

Solution: Single word alignment means 4-byte alignment. The header requires 4 bytes, and the footer is a duplicate, so it requires the same amount. Since zero-size payloads are not allowed, the smallest valid request is for 1 byte. To satisfy 4-byte alignment we would need 3 bytes of padding, making the minimum 12 bytes for both allocated blocks, whereas free blocks only need to be 8 bytes in size.

Still using single word alignment, but this time without a footer for allocated blocks, we would require a block of at least 8 bytes to satisfy the request for 1 byte; this is the same size needed for a free block.

For a double word alignment and allocated blocks with block header and footer, we need 8 bytes for header and footer, 1 byte for the request, and 7 bytes of padding to satisfy alignment. Thus the minimum block size would be 16 bytes.

Still using double word alignment, but with only a header for allocated blocks, we would have a 4-byte header, 1 byte bot the smallest request, and 3 bytes of padding for alignment. Thus the smallest allocated block would be 8 bytes.

Alignment	Allocated block	Free block	Minimum block size (bytes)
Single word	Header and footer	Header and footer	12
Single word	Header, but no footer	Header and footer	8
Double word	Header and footer	Header and footer	16
Double word	Header, but no footer	Header and footer	8

Exercise 9.8. Implement a `find_fit` function for the simple allocator described in Section 9.9.12.

```
static void *find_fit(size_t asize);
```

Your solution should perform a first-fit search of the implicit free list.

Solution: First fit search, as described in Section 9.9.7, searches the free list from the beginning and chooses the first free block that fits. For the memory model in question, every block, whether free or allocated, has a 4-byte header encoding the size of the block as well as a single bit indicating whether it is allocated or free. The useful data provided to the caller is immediately past the header, and it is referred to as the block pointer. For free blocks, there is also a footer, which is a copy of the header.

The list begins with a 4-byte padding, followed immediately by the prologue, which is an allocated block with just a header and a footer, and zero-sized payload. The header of the prologue encodes a size `DSIZE` and it always remains allocated. The `head_listp` is a block pointer that always points to the prologue block, and our search begins there. We can use the `HDR` macro on the block pointer, chained with the `GET_SIZE` macro to get its size. Since the heap list is terminated by an epilogue header block whose size is encoded as 0 and is always allocated. Thus, we continue our search as long as the current block has a nonzero size.

```
static void *find_fit(size_t asize)
{
    void *p;
    for (p = head_listp; GET_SIZE(HDRP(p)) != 0; p = NEXT_BLKP(HDRP(p)))
        if (!GET_ALLOC(HDRP(p)) && GET_SIZE(HDRP(p)) >= asize)
            return p;
    return NULL;
}
```

Exercise 9.9. Implement a `place` function for the example allocator.

```
static void place(void *bp, size_t asize);
```

Your solution should place the requested block at the beginning of the free block, splitting only if the size of the remainder block would equal or exceed the minimum block size.

Solution: The procedure is used by `mm_malloc` once a suitable block has been found. It updates the header on the block found by turning on the allocation bit in the header and compares the requested size against the actual size of the block. In the event the block's size is bigger than the requested size, and that the remainder size is enough to allocate a header, a footer, and at least 1 block of useful data (in total `2 * DSIZE`), it writes a header and footer just past the current block being allocated with the remainder size, setting the allocated bit to 0.

```
static void place(void *bp, size_t asize)
{
    size_t total = GET_SIZE(HDR(p));
    size_t remaining = total - size;
    PUT(HDRP(p), PACK(asize, 1));
    PUT(HDRP(p), PACK(asize, 1));
    if (remaining >= 2 * DSIZE) { /* Enough for header, footer, and useful data */
        /* Make caller's block just big enough */
        PUT(HDRP(bp), PACK(asize, 1);
        PUT(FTRP(bp), PACK(asize, 1);
        bp = NEXT_BLKP(bp);

        /* Assign remainder to free list */
        PUT(HDRP(bp), PACK(remaining, 0));
        PUT(FTRP(bp), PACK(remaining, 0));
    } else {
        /* Assign entire thing to caller */
        PUT(HDRP(bp), PACK(total, 1));
        PUT(FTRP(bp), PACK(total, 1));
    }
}
```

Exercise 9.10. Describe a reference pattern that results in severe external fragmentation in an allocator based on simple segregated storage.

Solution: Recall that, as described in Section 9.9.4 on page 846, *external segregation* occurs when there *is* enough aggregate free memory to satisfy an allocate request, but no single free block is large enough to handle request.

In simple segregated storage, the free list for each size contains same-size blocks, each the size of the largest element of the size class. Moreover, because blocks are never coalesced. If requests come in for blocks larger than the size of any one class of blocks, then this scheme will request memory from the operating system each time, even though there is enough

aggregate space. For example, if there are several blocks with 2048 bytes in size, and the request is for a block of size 4096, then the memory system will request memory from the operating system, and create a list of blocks of the requested size before handing us one. Then, another request comes in for a size double that one, and the pattern repeats, even though there certainly is enough memory to fulfill it if we were to coalesce the free blocks.

Exercise 9.14. Given an input file `hello.txt` that consists of the string `Hello, world\n`, write a C program that uses `mmap` to change the contents of `hello.txt` to `Jello, world!\n`.