Operating systems and C

Exam answers

IT UNIVERSITY OF CPH

Andreas Nicolaj Tietgen
anti@itu.dk

16. Dec 2022

Contents

1 Data lab

1.1 Describe your implementation of `howManyBits(x)`

The 'how many bits' task is about counting the minimum amount of bits required to represent a specific integer. Sadly I did not finish this in time for the hand-in deadline. However, I have figured a way to solve the problem. The function performs a binary search for the most significant active bit. The issue is that we also need to do this for negative numbers where the sign bit is active. In order to solve this we then take the bitwise negation of x. Doing so leaves us with a number one short of being the actual absolute value. We do not want the actual absolute value since there is a risk of overflowing back to a negtive number. In order to know how many bits that is needed to represent the integer, we count it at each step. The binary search algorithm starts in the middle of the bit representation, as the line indicates in figure 1. If there is an active bit on the left side, we can skip what is on the right side. That means the amount of bits that we do not need to look at on the rigth side will be added to the result as the minimum amount of bits needed to represent the integer. When the binary search is finished, we add 1 to the result so that we remember the sign bit.

As an example we have `x = 0xA0FFF0F0` which is a negative number. We find out that it is a negative number by looking at the signed bit. We then take the logical negation of the number and assign the value to the `temp`, so that `temp = 0x5f000f0f`. Now what we want to do is to figure out where the most significant active bit is. As mentioned, it can be found by performing binary search.

As the line indicates in figure 1, we start from the middle:

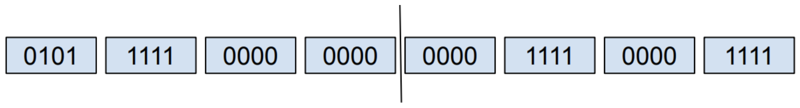

Figure 1: Bit representation of the negated integer of `x`

We look to the left of the line and check if there is at least 1 bit that is active(i.e, a bit with the value 1). In figure 1, there is a bit that is active. For now, we can with confidence say that we at least need 16 bits in order to represent this number. This is due to the active bit in the left side, so we do not need to look at the right side.

The following code provides the same result:

```c
    int active_bits_left = !!(temp >> 16);
```

```
2      min_required_bits = active_bits_left << 4;
3
4      /*
5      * Shift the result such that we can continue the search for
           the
6      * last active bit
7      */
8      temp >>= min_required_bits;
9      result += min_required_bits; // Store the min required bits
```

The `!!(temp » 16)` looks at the left half of the bit representation and returns 1 if there is an active bit and 0 if not. We then bit shift it by 4 in order to get the amount of bits that we can guarantee that is required to represent the integer.

At line 8 we bit shift the `min_required_bits`, i.e. 16 in this case, such that we can continue our search. The result looks like the following:

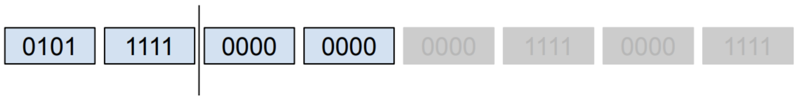

Figure 2: The bit representation of temp after bits shifting it. Grey blocks have been bit shifted out

As figure 2 shows, we now continue the process of binary search and take the half again. Again we can see that there is a bit on the left side of the line. This guarantees that we now at least need 8 bits on top of the previous 16 bits that we have, i.e. 24.

The code to perform this looks nearly the same as in figure 1:

```
1      int active_bits_left !!(temp >> 8);
2      min_required_bits =  active_bits_left << 3;
3
4      /*
5      * Shift the result such that we can continue the search for
           the
6      * last active bit
7      */
8      temp >>= min_required_bits;
9      result += min_required_bits; // Store the min required bits
```

The difference is at line 1 and 2. At line 1 we half the amount of bits that we want to bit-shift with at each step until it reaches zero. And at line 2 we decrement the amount of bits we bit shift the result of the `active_bits_left` by one at each step until it reaches zero. Another thing to notice as well is that we continue to add the minimum required bits that we bit-shift out of the `temp` at each step. The following figure is an illustration of continuing the steps described:

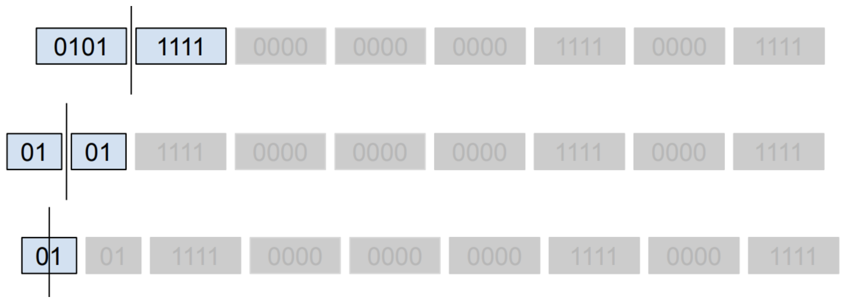

Figure 3: An illustration of the continuation of the same steps. The grey blocks have been bit-shited out

Before the last step of figure 3, the minimum required bits is 30. The last step is a bit different. The left side of the line does not have an active bit. For that reason, the variable `minimum_required_bits` is 0 and we do not bit shift. The process of performing binary search is done. However we still need to add the result of `temp`. Because if the bit on the right side of the line is active, then that needs to be added. Due to our binary search and bit-shifting of the binary representation of the integer, then we only add 1 or 0 to our result. Before we return the result, we need to add the sign bit. For that reason we add 1 to the result. We return a result of 32 minimum required bits to represent `0xA0FFF0F0`.

1.2 Describe your implementation of `tmin(void)`

The `tmin()` task was about creating the minimum number in a two complement bit representation. Two's complement uses the most significant bit as a sign bit. That is, the bit at the left most position indicates whether it is a negative or positive number. If the bit is 1 then the number is a minus. The minimum number in a two's complement system is represented by having the most significant bit set to 1 and then the rest of the bits set to 0. In a 32 bit system it would look like the following:

$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = -2147483648_{10}$$

Following the set of rules for the assignment, it is solved by having a constant which is `0x01` and then bit-shift by 31 positions like the following:

```c
    int tmin() {
        return 0x01 << 31;
    }
```

2 Perf lab

2.1 A. What is the difference between spatial and temporal locality

Spatial locality is when we access something in memory then it is likely that we reference some of the nearby addresses some time after. An example is when we loop through arrays. Then we take the next element each time.

```c
void print_items(int a[N]) {

    for(int i = 0; i < N; i++) {
        printf("%d \n", a[i]);
    }
}
```

As can be seen above, the loop goes through the elements of the array, by accessing it one by one. Here we get the elements by utilizing how the array structure the data. That is, we get the next item in memory which is from a nearby memory address. *Temporal locality* is when an item that is being referenced is likely to be referenced again in the near future. An example of temporal locality is when we reference a specific variable at each iteration.

```c
int pow(int a, int pow) {
    int tmp = a;

    for(int i = 0; i < pow; i++) {
        tmp = tmp * a;
    }

    return tmp;
}
```

Spatial and *temporal locality* is very important to limit the amount of cache misses. This is very important in terms of performance. The better the application utilizes the cache, the faster the application performs. When an application needs a value it first checks if the cache has the value that we are looking for. If not, it goes further down the memory hierarchy, to get the data that the application needs. Since the cache is the closest memory to the CPU then it is also the fastest for the application to use. The difference between main memory and the L1 cache is around 100 clock cycles. So by utilizing our cache and limit the cache misses with spatial and temporal locality, our applications run faster.

2.2 B. What is SIMD processing

SIMD, also known as *Single Instruction Multiple Data*, is a way to perform computations on a lot of data with one instruction. The way that it works is by defining a vector with values that we want to perform computations upon. The `gcc` can generate SIMD with AVX instructions by adding the flag `-mavx2` (Randal E. Bryant 2016b).

My solution does not utilize SIMD. That is due to not using vectors that we can perform arithmetic operations upon. However, the smooth solution could have benefitted from it in the cases where we need to take the average of $3 * 3$ pixels. Here we could have created an accumulator vector and a vector for each row and use the `+` between the accumulator and the row. Afterwards we can add each value in the accumulator vector and divide it by nine. This will lead to removing 3 add instructions in the `avg_smooth` function, making it more performant.

3 Malloc lab

3.1 Explain in detail your implementation of the `mm_malloc` function

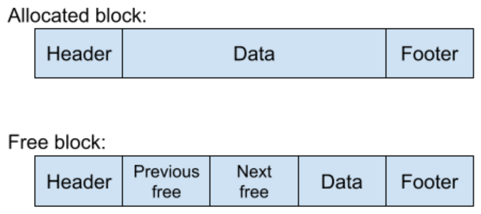

Figure 4: Show how an allocated and free block is structured

In order to understand the implementation of `mm_malloc()`, then we have to understand how we keep track of the free memory blocks that my dynamic memory allocator manages. In figure 4 is a representation of an allocated and a free block. The header and footer contains the same data i.e., the size and a flag indicating if it is allocated or not. The difference between the allocated and the free block is that we have a previous free and a next free block pointer. We use these pointers, to link our free blocks together into an *explicit free list*(See figure 5).

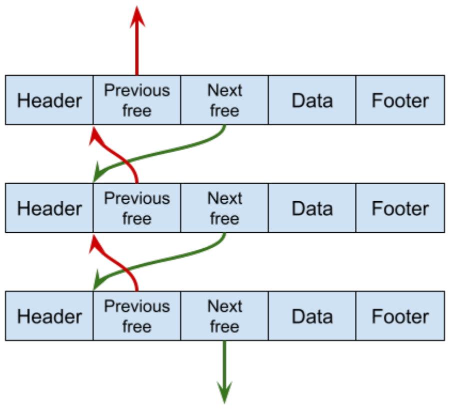

Figure 5: shows how explicit free list is structured

The explicit free list only points to free blocks. For that reason, we use the previous and next free block pointers to create a linked list. When we insert a new block to the free list, we insert it at the beginning. We set the free pointers such that the new free block points with the next pointer to the previously first free block in the list and by making the previously first free block setting the previous free block pointer to point to the new free block.

However, the implementation of the explicit free list contains every free block, unordered in size. The potential running time of getting a free block, that is equal or greater than the size we want is $O(n)$, where n is the number of free blocks in the explicit free list. For that reason, I have implemented segregated free list. As figure 6 shows, it consists of an array of pointers to explicit free lists. Each entry of explicit list only contains particular sizes.

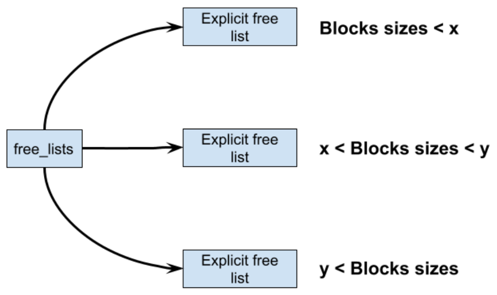

Figure 6: Segregated free lists consist of an array of explicit free lists, where the explicit free list have its own equivalence class

This allows us to skip free blocks that are too small and look in the explicit free lists that could have a free block that fits in size.

The implementation of the `mm_malloc()` function, is the same as the books implementation(Randal E. Bryant 2016a, ch. 9.9.12). It first checks the requested `size`. If the size is equal to 0, we have to return `NULL`. Afterwards, we have to adjust the variable `size` to be aligned to 8 and save it to a variable called `asize`. There are two cases that needs to be taken care of:

- When the requested size is below or equal the constant `DSIZE`: We set the adjusted size, i.e. `asize`, to $2 * DSIZE$.
- When the requested size is above the constant `DSIZE`: We use the `ALIGN()` macro with $size + DSIZE$ as an argument. We add `DSIZE` to the size as input in `ALIGN()` to ensure that we have space for the header and footer.

After adjusting the size, we use the `find_fit()` function to find a free block that can contain the adjusted size, `asize`. If it returns `NULL` then we need to extend the heap. When we have obtained a free block, then we need to allocate it. `place()` takes care of allocating the free block, and returns a pointer to the allocated block.

3.1.1 `find_fit()`

The `find_fit()` utilizes the segregated free list implementation. This can be seen in figure 7 where it gets the index of the first list that could contain the size being requested. This ensures that we skip the lists that do not

```
int index = get_free_list_index(size);
```

Figure 7: Shows Line 4 in appendix B

have sizes that fit the requested size. The index serves as a starting point of where to look for a free block. The function goes through each explicit free list from the starting point(see line 6-8 in appendix B). Due to the first fit implementation it will stop as soon as it finds a free block that fits the requested size. If the current list, that we go through, points to `NULL` or does not have a block that fits, we increment the index by 1 and get the next list. If `find_fit()` is not able to find a free block that fits, it will then return `NULL`.

3.1.2 `place()`

Before `place()` starts allocating the free block, it first calculates the difference in size between the size of the free block and the requested. The result of the calculation is stored in `size_diff`. Afterwards it removes the free block from the free list by calling the `remove_free_block()` function. We do so to ensure that the free block we want to allocate, is not going to be present in the free list anymore. Now that it is removed, the function then decides whether we should split the block which we want to allocate. `place()` is going to split if the variable `size_diff` is greater than $2 * DSIZE$. We need it to be greater than $2 * DSIZE$ such that we have place for the header, footer, next pointer, and previous pointer.

If the function decides to split then it does the following(see line 26-32 in appendix C):

- Sets the header and footer of the block we want to allocate, by setting the size to be equal to the `asize` and allocation tag to be 1.
- We then jump to the next free block, with `NEXT_BLKP` and sets the header and footer to be the size to `size_diff` and the allocation tag to 0, i.e. free.
- Lastly it adds the new free block to the free list with the function `insert_free_block`.

If the `place()` function decides not to split, it allocates the free block by setting the header and footer with an allocated allocation flag, but leaves the size as it is, like the following code:

```
PUT(HDRP(bp), PACK(size, 1));
PUT(FTRP(bp), PACK(size, 1));
```

3.2 What is pointer arithmetic?

Pointer arithmetic is a way to move the pointer to another virtual memory address. The special thing to remember is that the amount of bytes that the address moves is depending on which type of pointer it is e.g. `char`, `integer`, `long`, and etc.

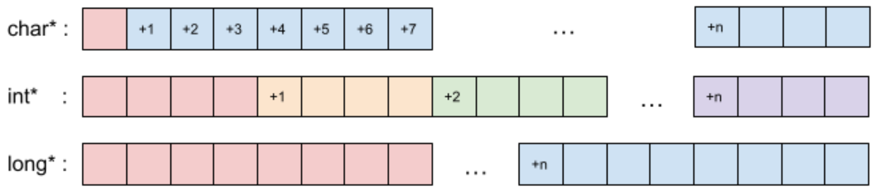

Figure 8: Show the difference when adding n to a pointer of a specific type. Each block represents one byte. When adding 1 to a char pointer it moves only by one byte. When adding 1 to an int pointer it moves by 4.

As figure 8 shows, in a 64-bit word system, an `int*` called `int_p` and a `char*` called `char_p` would move at their own respective data-size. That means that `int_p+1` would move 4 bytes, whereas the `long_p+1` would move 1 byte. As an example, in malloc lab we move a pointer only 1 byte by casting the pointer to a char pointer and add one just like:

`HDRP(bp) ((char *)bp - WSIZE)`

4 Topics from the class

4.1 A. What is the difference between traps, faults and aborts in the context of interrupts?

Traps, faults and aborts are different exceptions that happens in an operating system.

A *trap* is an intentional exception. What happens is that a system call triggers the *trap* and passes control to an exception handler. The exception handler then handles the trap and returns to the instruction that is after the system call. *Traps* are typically used for calls that request services from the kernel. This could be reading a file with the system call `read` or creating a new process with `fork`.

A *fault* happens when an instruction causes an error. When this occurs, it then passes control to an exception handler for that specific fault. Here two things can happen. The exception handler can handle the fault and give back control to the application by re-executing the instruction. It could also choose to abort instead and terminate the application that caused the fault.

An *abort* is when the function reaches a fatal error. Again, we pass control to an exception handler. However, the exception handler does not try to repair the situation. Instead it is being sent to the abort routine never giving control back to the application.

4.2 B. What is the difference between an ephemeral and a well-known port?

The ephemeral ports are something that is being assigned automatically by kernel of the client. The operating system has a predefined range of ports that it uses to automatically provide a port when a client application wants to communicate through. On the other hand, well known ports are typically associated with some service and has been agreed upon by the community. As an example the port 80 is a well known port for the *http* protocol.

4.3 C. What is a memory leak?

A memory leak is when something that is allocated is never cleaned up while the application is running. It is a lethal thing for an application to occur. This normally happens when we allocate some data in the heap by using `malloc`, from the standard library, and forgets to use the function `free` to clean up the memory at that specific location.

```c
    int do_something() {
        int *numbers_p = malloc(sizeof(int)*10);

        // Init numbers
        for(int i = 0; i < 10; i++) {

            numbers_p[i] = i + 1;
        }

    int sum = get_sum(numbers_p, 10);

    return sum;
}

int get_sum(int *numbers, int count) {
    int sum = 0;

    for(int i = 0; i < count; i++) {

        sum += numbers[i];
    }

    return sum;
}
```

We can see from the above code that we allocated 10 integers in the heap and initialize them to $1, 2, 3...10$. Then the `get_sum()` calculates the sum of all the values and returns it. At the end of function `do_something()` it returns the `sum` value. However, the `numbers_p` still remains in the heap since we have not called the `free()` function.

Having a memory leak in your application indicates that it forgets to clean up. In order to avoid memory leak we need to keep track of what we have allocated and when we would like to clean it up with the function `free()`.

4.4 D. What is a race-condition?

A race condition can happen when two concurrent processes change the same variable at nearly the same time. More specifically this can happen because thread A access the variable `int a`. Before the thread begins to add 1 then the operating system performs a thread context switch and then thread B reads the same variable and write `a + 1` to it. However, thread A didn't get those changes due to reading the variable before it gave the control to thread B. Now thread B reads variable and adds 1 to it. Since thread A still have the old value then it adds 1 to the same value. So for that reason the variable is only added by 1 instead of 2 in this example.

This specific scenario can be very hard to debug. But why? Race conditions does not always occur. So if you were to try debug and recreate the example then another example could occur or it might actually work as expected(in that specific run). We can use the concept of a *mutex* or a *semaphore*. These two provide operations that, if used correctly, can ensure that only one thread at a time can access the critical section at a time.

It is expensive because even though we have secured the critical section, then all of the threads have to wait until it gets access to the critical section. In other words, they cannot do any work until it is being let through. This can cause some bottle necks in terms of gaining performance if something is taking long time to compute in the critical section.

References

[Ran16a] David R. O'Hallaron Randal E. Bryant. *Computer Systems - A Programmer's Perspective*. Pearson, 2016. ISBN: 978-0-13-409266-9.

[Ran16b] David R. O'Hallaron Randal E. Bryant. *CS:APP3e Web Aside OPT:SIMD: Achieving Greater Parallelism with SIMD Instructions*. Accessed: 14-12-2022. 2016. URL: `http://csapp.cs.cmu.edu/3e/waside/waside-simd.pdf`.

Appendix

A mm_malloc() Implementation

```c
/*
 * Allocates a block of memory with the requested size.
 *
 * Input:
 * size - The size that should be allocated
 *
 * Returns:
 * The pointer to the allocated block of memory.
 * If it is not able to allocate more memory then it returns NULL
 */
void *mm_malloc(size_t size)
{
    size_t asize; /* Adjusted block size */
    size_t extendsize; /* Amount to extend heap if no fit */
    char *bp;

    /* Ignore spurious requests */
    if (size == 0)
        return NULL;

    /* Adjust block size to include overhead and alignment reqs.
        */
    if (size <= DSIZE)
            asize = 2 * DSIZE; // Sets size to 2 * DSIZE (for
                header and footer)
    else
        asize = ALIGN(size + DSIZE);

    /* Search the free list for a fit */
    if ((bp = find_fit(asize)) != NULL) {
        place(bp, asize);
        return bp;
    }

    /* No fit found. Get more memory and place the block */
    extendsize = MAX(asize,CHUNKSIZE);
    if ((bp = extend_heap(extendsize/WSIZE)) == NULL)
        return NULL;

    place(bp, asize);
    return bp;
}
```

B `find_fit()` implementation

```c
static void *find_fit(size_t size) {

    // Get index for free list based on size
    int index = get_free_list_index(size);

    for(int i = index; i < num_free_lists; i++) {

        char *current = free_lists[i];

        if(current == NULL)
            continue;

        // Go through the list and take the first that fits
        while(current != NULL) {

            size_t current_size = GET_SIZE(HDRP(current));

            /* If the size of the current free block is greater
                than the requested
            then return it */
            if(current_size >= size)
                return current;

            current = NEXT_FREE_BLOCK(current);
        }
    }

    return NULL;
}
```

C `place()` implementation

```c
/*
 * Place the block of memory by setting the header and footer
 * to be allocated with the specified adjusted size.
 *
 * If the given block pointer has a size difference greater
 * than or equal to 2 * DSIZE then it is going to split the
 * block and insert the rest of the splitted block into the
 * free list
 *
 * Inputs:
 * bp - pointer to the block of memory that should be placed
 * asize - the size that should be placed
```

```c
*/
static void place(void *bp, size_t asize) {
    size_t size = GET_SIZE(HDRP(bp));
    size_t size_diff = size - asize;
    unsigned int should_split = size_diff >= 2 * DSIZE;

    remove_free_block(bp);

    if(!should_split) {

        PUT(HDRP(bp), PACK(size, 1));
        PUT(FTRP(bp), PACK(size, 1));
    } else { // Split bp and inserted the rest into the free list
        PUT(HDRP(bp), PACK(asize, 1));
        PUT(FTRP(bp), PACK(asize, 1));

        void *split_p = NEXT_BLKP(bp);
        PUT(HDRP(split_p), PACK(size_diff, 0));
        PUT(FTRP(split_p), PACK(size_diff, 0));
        insert_free_block(split_p);
    }
}
```