SID-LAKHDAR Riyane
M1 MoSIG

Operating Systems

Practical session 2

Abstract :

Attached to this report, you will find the next elements :

- src : directory containing the source code corresponding to the program without « safety check ».
- src_safety_check : directory containing the same program with the safety check asked in the question IV
- src_fragmentation : directory containing the same program with a feature that allows the user to have a track of the fragmentation performences of the allocator.
- compileAndRunTest.sh : a script that compiles all the source codes. Then it runs the program with all the input test file. The output file of this executions is compared with the corresponding results. The diffrence between the result for each input file and the corresponding output file is printed.
- printFragmentationTest.sh : a script that runs sevral programs (ls, ps, ...)with our librabys on specific parameters to allow him to track the fragmentation value. The corresponding plot is put in the directory « output/plot »

III. Allocator Implementation and Validation

Question III.1

The free block structure provides us a list of the contiguous free blocks of the memory, and the size of each block. The algorithms we have implemented try to keep this list sorted from the lower address to the biggest one. So to find out a busy block bb, we just need to go throught this list to find the two free blocks on the side of our busy block. Which gives us a list of contiguous busy blocks. Then we go through thous busy blocks (as we know them size) till we find (or exceed) the expected busy block bb.
Thus, we do not need to keep the list for occupied blocks.

Question III.2

No matter which allocating algorithm we are using, the 1, 2 or 3 addresses will be used as metadata by our system to store the header of the first busy block (or free one).
However, at writing time, no check is performed by the system to insure that the given address belongs to a proper busy block.

Question III.3

A busy block is composed of two contiguous parts :

- The header of the block (struct busy_block).
- The memory space dedicated to the user purposes.

Thus, when a block is allocated at the address a, the user should receive the address :

a + sizeof(struct busy_block) which is the address of the first useable byte.

Question III.4
When the user deals with memory addresses, the only addresses he knows are the addresses of the first memory space dedicated to his purposes (and not the header addresses).
Thus, to free a memory block, the system must be able to use this address to find out the address of the header of the block.
Then he needs to know the addresses and sizes of the free block on its sides to eventually merge the new free block with them (if contiguous).

This specific cas appears when the diffrence between the founded free memory and the expected size is smaller the the free block structure size.
To deal with this case, we can use internal fragmentation : We allocated the hole free block.

IV. Integrating Safety Checks:

Some of the safety check described in this section have already been done in the previous version of our code :

- **Forgetting to free memory :**
  As the system can not force the user to make any free, the only way to warn the user that he is maybe making some memory leaks is to print somme errors or return specific values when he is creating a memory overflow. Thoes too solutions have been implemented in our previous version of the program.
- **Corrupting the allocator metadata :**
  To detect data corruption, we have determinated, within our allocation and free algorithme, some cases that shoul never happen in a good program behaviour.
  Eg : `two overlapped free memory blocks, busy and free block…`
  In each of this cases, a specific error is printed and the program exits.

The following safety check has been implementd in the new vertion of the program

- **Calling free() incorrectly :**
  When the user calls our free function, the first action is to check if the guiven address corresponds to the first byte of an allocated memory block. To do so, the program finds the two free block surounding the given adress. Then it looks for all the busy blocks within this gap to see if the given address corresponds to the bigining of a busy block.
  If this test fails, the corresponding error is printed and the program exits. This strategy allows us to avoid freeing partial busy blocks, or to corrupt meta dattas.

V. Measuring Fragmentation

1. External fragmentation

To evaluate the fragmentation of our memory, and compare the fragmentation of our different allocation algorithm, we have designed an optional feature to our allocator.
In this version of the program (directory « src_fragmentation »), each time an allocation or free function is called, the fragmentation is recomputed and printed into some specific file.
To compute this fragmentation, we have designed two different computing function :

- The first way is to compute the fragmentation value as :
  sum for each free block(1 / size of the free block).
  Thanks to this formula, the smallest the free block size will be, the biggest fragmentation will be. Thus we can carecterise the external fragmentation of the mermory are inversly proportional to the free block size and them number.
  Thanks to this method, we have been able to observe the external fragmentation of our allocator while runing on sevral programs like ps and ls.
- An other way to caracterise external fragmentation is to observe the distribution of the free block regarding them sizes :
  We have ploted a historgam which represent for each free block size the number of block with the given size. Thos, the more small block will be numerous, the more the fragmentation will be inportant.

The result of our different algorithms using each of the fragmentation methods are given in the directory « output/plot ».

2. Internal fragmentation

To compute the internal fragmentation, we wanted to design the same feature of our program. The only difference concerns the evaluation function:
Every time the memory block returned to the user is bigger than the asked size (regarding less to the block header), the internal fragmentation will increase.
Each time allocated block is removed, the internal fragmentation will decrease.
To do so, we appended to our busy_block structure an integer which indicates the size of the additional memory allocated.

VI. Conclusion

During this lab, we have designed and implemented a memory allocator using different allocation polycies. This allocator may use sevral feature to ensure a safety behaviour, or to track its performances.
We have been able to develop different test. Our program has been stressed within a specific test environement, and in a more general purposed environement.
Finally, the performance track we have designed allowed us to see the adventages and desadventages of each allocation strategy.