Segmenting a memory

Johan Montelius

HT2021

1 Introduction

In this exercise you will create a small framework to handle allocation of arrays. The way we solve it is a bit over-kill since we could have used `malloc()` and `free()` directly but to learn a bit about segmentation it serves its purpose.

The scenario we have is that we will be given a `memory` that will be used when an `array` is created. The array will have its own header structure with information about its size etc but the content of the array will be in the shared memory.

When a new array is created we need to find a segment in the memory that should be allocated to the array. In order to do this we need to keep track of which arrays that have been created and which segments in the memory that they use.

In the beginning the arrays we create will simply take memory locations after each other. Once we start to delete arrays the picture becomes more complicated. The memory will become fragmented with segments that are still used and between them areas that are free to grab.

If the memory system is asked to create an array but can not find any memory area that is big enough we have a problem. We will then try to solve it by compacting the segments that are in use.

2 Memory and arrays

Let's start with defining a small memory area and an array data structure. The array structure will hold the size of the array and a pointer to its first position in the shared memory. It will also hold a pointer to the next array structure since we will keep track of all arrays that we create in a linked list.

```
#define MEMORY 100          // size of memory

int memory[MEMORY];         // the memory

typedef struct array {
  int size;                 // the size of the array
  int *segment;             // pointer to the allocated memory segment
  struct array* next        // pointer to next array
} array;
```

We could have simply defined one global variable `allocated` that would either be a null pointer or pointing to the first element in the linked list of allocated arrays. We would then always have some special cases in our code to detect if the list was empty or if we are looking at the last element. Therefore we will make use of two faked elements, the first one called `dummy` and the last one `sentinel`. So when we set up the initial list we let `dummy` point to `sentinel` and the variable `allocated` point to `dummy`.

```
array sentinel = {0, &memory[MEMORY], NULL};
array dummy = {1, &memory[-1], &sentinel}
};

array *allocated = &dummy;
```

The dummy array element has a size of one and its memory segment starts at `memory[-1]`. This is of course crazy wince we do not want to store anything at this position but we will never use this array. The important thing is that we can calculate the next, possibly free, memory location by taking the address of the segment and add the size of the array. We would then point to `memory[0]`.

The next element in the list is the sentinel and its memory segment starts at `memory[MEMORY]` that is one position outside the actual memory data structure (remember that arrays in C are 0 indexed so an array of size $MEMORY$ would run from zero to $MEMORY - 1$). This is also very strange but we ill never use the sentinel array either.

The trick is that we can now look at the first element and the following one (i.e. the sentinel) and determine that the memory area from `memory[0]` to `memory[MEMORY -1]` is free to use. This is an invariant we will uphold in the list; by looking at two consecutive elements we can determine if and how much free memory there is between them.

Sound strange? You will understand how it works when we use this property.

3 Operations on the list

Let's start with a procedure that will help you see what is happening in the list. We run through the list and print information about each element. Notice how we print the segment pointer, by subtracting `memory` they are much easier to interpret and compare to the sizes.

```
void check() {
  array *nxt = allocated;
  while(nxt != NULL){
    printf("array (%p) : size %2d, segment %3d, next %14p \n",
           nxt,
```

```
            nxt->size,
            nxt->next,
            (int)(nxt->segment - memory)
          );
    nxt = nxt->next;
  }
  return;
}
```

You should already now write a small main procedure and compile the program you have so far. It's easier to detect the bugs as you go instead of waiting to the end and the start to correct everything.

```
int main() {
  check();
  return 0;
}
```

You should then implement a procedure that allocates a new array. We do it step by step since this is the most complicated code. The overall structure look like this:

```
array *allocate(int size) {
  array *nxt = allocated;

  while(nxt->size != 0){
    if((nxt->next->segment - (nxt->segment + nxt->size)) >= size) {
        :
        :
    }
    nxt = nxt->next;
  }

  return NULL;
}
```

When we're give a size and are asked to allocate an array we run through the list starting at `allocated` until we hit the sentinel. We know that it is the sentinel since its size is zero. If we hit the sentinel without finding room for the new array we return NULL.

To determine if there is room for a new segment we look at the current array (`nxt`) and its successor (`nxt->next`). We then apply the trick of inspecting the segment addresses to calculate the free space in between the segments. If this space is equal or larger then size then we have found room for the segment we need.

So what do we do when we find room for a new array; the first thing we do is allocate a new array structure on the heap using `malloc()`. We then

initialize the fields and insert the new element in the linked list and return the array.

```
        array *new = (array*)malloc(sizeof(array));

        new->size = size;
        new->segment = (nxt->segment + nxt->size);
        new->next = nxt->next;
        nxt->next = new;
        return new;
```

You might now wounder if we take care of the case that the list is empty or we should add a new entry in the end of the list bu this is where the dummy and sentinel elements saves us. We will always insert a new block between two existing blocks.

We will also use a wrapper procedure called `create()`. This procedure will simply call `allocate()` and print an error message if an allocation could not be made.

```
array *create(int size) {
  printf("create an array of size %4d ..", size);
  array *new = allocate(size);
  if(new == NULL) {
    printf("out of memory\n");
    exit(-1);
  }
  printf("done\n");
  return new;
}
```

The next procedure we need is a procedure that removes an allocated array. To do this we simply run through the list and search for the element in question. Since we know that we will never remove the first nor the last element the code becomes quite straight forward:

```
void delete(array *arr) {
  printf("delete array (%p) of size %4d", arr, arr->size);
  array *nxt = allocated;
  array *prev = NULL;

  while(nxt != arr){
      :
  }
      :
      :
  printf("done\n");
  return;
```

```
}
```

Fill in the blanks, unlink the array and don't forget to `free()` the array data structure.

The two last procedures are to read and write values to an array. belts. Like sensible people we want our high level arrays to be one indexed. Let's keep it simple to begin with and run without safety:

```
void set(array *arr, int pos, int val) {
  arr->segment[pos-1] = val;
}

int get(array *arr, int pos) {
  return arr->segment[pos-1];
}
```

Mission complete, lets take it for a spin.

4 A first test

Let's write a small test to see that we can work with our arrays. We will create two arrays, do some operations and then delete them.

```
void bench1() {

  check();
  array a* = create(20);

  check();
  array b* = create(30);

  check();
  set(a, 10, 110);
  set(a, 14, 114);

  set(b, 8, 208);
  set(b, 12, 212);

  printf(" a[10] + a[14] = %d\n", get(a,10) + get(a, 14));
  printf(" b[8] + b[12] = %d\n", get(b,8) + get(b, 12));

  delete(a);
  check();
  delete(b);
  check();
}
```

If this works we can do something more complicated. Let's see if we can find room for a small segment in between two segments. We will first create three arrays, delete the middle one and then create a new array.

```
void bench2() {

    array *a = create(20);
    array *b = create(30);
    array *c = create(30);
    check();
    delete(b);
    check();
    array *d = create(20);
    check();
    delete(a);
    delete(c);
    delete(d);
}
```

5 Base and bounds

So far we do not check if the set and get procedures are valid operations. We can do very strange things as the following example shows:

```
void bench3() {
    array *a = create(20);
    array *b = create(30);

    set(a, 22, 100);
    set(b, 0, 200);

    printf("a[20] + b[2] = %d\n", get(a,20) + get(b, 2));

    delete(a);
    delete(b);
}
```

This example shows the danger of not checking the bounds of the allocated segment. Change the implementation of `set()` and `get()` so that they check that the position is a legal value i.e. not larger than the size of the array nor zero or smaller.

If this is the case you should print an error message stating that an "segmentation fault" has occurred and then call `exit(-1)`. When you program in C you will often see a message "segmentation fault (core dumped)" and

this is exactly what is happening, your trying to address a location that is not inside the segments that have been allocated to your program.

6 More problems

One thing we could take a look at is a shortcoming of our current implementation. Take a look at the following benchmark:

```
void bench4() {
    array *a = create(20);
    array *b = create(30);
    array *c = create(30);
    delete(b);
    check();
    array *d = create(50);

    delete(a);
    delete(c);
    delete(d);
}
```

We fail but do we have to fail? If you look at the segments you see that we have one segment allocated at position 0 of size 20 and another one at position 50 of size 30. We thus have 20 slots free between the two segments and 30 slots free after the second segment. With 50 free the memory is hardly full, the problem is that it is fragmented (*external fragmentation*, areas that have not been allocated but all too small to be used).

We can try to fix this problem by moving the second segment closer to the first segment. This would free up 50 slots after the second segment and give room for the third array. We implement this by doing a *compaction* of all segments that we have allocated.

A compaction is actually quite easy to do. We know that the first segment is the dummy segment so this should not be touched but for all other segments apart from the sentinel we move its content closer to its predecessor.

```
void compact() {

  array *prev = allocated;
  array *nxt = prev->next;

  while(nxt->size != 0) {
    for(int i = 0; i < nxt->size; i++) {
      prev->segment[prev->size + i] = ...   ;
    }
```

```
    nxt->segment = .....   ;
    prev = nxt;
    nxt = nxt->next;
  }
}
```

Fill in the dotted lines; think about what needs to be done. You should read each element in an array and place it immediately after the segment of the previous array. When you're done you update the segment pointer to point at the first element at the new position.

Once you think you have this working we change the implementation of the procedure `create()`. How about this:

```
array *create(int size) {
  printf("create an array of size %4d\n", size);
  array *new = allocate(size);
  if(new == NULL) {
    printf(".. almost panic, time for gc\n");
    compact();
    check();
    new = allocate(size);
  }
  printf(".. pray for the best ..");
  if(new == NULL) {
    printf("panic! memory full\n");
    exit(-1);
  }
  printf("... yes!\n");
  return new;
}
```

The compaction that we do is a form of *garbage collection* used in many programming languages such as Java, Haskell or Erlang. A general garbage collector is much more complicated since we need to move data structures that have references to other data structures. These should then also be moved but what if they already have been moved :-0