Page up and page down

Johan Montelius and Amir H. Payberah

1 Introduction

In this tutorial, you will implement an array allocation scheme that, on the surface, behaves very much like the array framework that you implemented in the segmentation exercise. This time you will use paging and try to understand why you first must recap the problems with segmentation.

The problem with segmentation is that you will end up with fragmented memory. Many small free memory areas together could make up a large part of the available memory, but they are all too small to be very useful. We could implement some compaction scheme, but this is not always easy, and you do not want to sit around waiting for defragmentation (remember Windows?).

To mitigate this problem, we instead use a paging scheme. You will see that it is a bit trickier, and if we did not have hardware support in the CPU, we would not be able to use it efficiently.

2 Memory and arrays

Let's start as before and define a memory we will use and what an array structure will look like. The general idea is that the memory will be divided into a sequence of *frames* and that each array will hold a *page table* that contains one entry per *page*. To make the system more flexible, we define two macros that determine how many frames we will have in memory and the *size* of pages/frames (they are of the same size since a page should fit into a frame).

```
#define FRAMES 64
#define SIZE 16
```

We have chosen 64 and 16 but could, of course, have chosen anything. The size is, however, preferably a power of two. The *virtual addresses* will be broken down into an *offset* (where in a page) and *index* (which page). Since the page size is 16, we need four bits as an offset, so we create a macro that does a bit-wise *and operation* to select the four least significant bits. Another macro will shift an integer four bits to select the higher bits as the page number.

```
#define Offset(addr) (addr & 0b1111)
#define PageNr(addr) (addr >> 4)
```

Note that to do these operations, we take for granted how an integer (that will be our address) is represented in binary. We also define two values, *FREE* and *TAKEN*, that we will use in a structure that keeps track of which frames are used. The memory itself is represented as an array of integers.

```
typedef enum available {FREE, TAKEN} available;
available framemap[FRAMES];
int memory[FRAMES*SIZE];
```

So now, to the actual array, the array is represented as an array or frame numbers, i.e., the page table. We also keep track of how large this structure is since we both want to check that we are not addressing outside of the array bit and also since we want to return all frames when we delete the array. The page table will use -1 to indicate that no frame has yet been allocated to the page.

```
typedef struct array {
  int pages;
  int *pagetable; // an array of frame numbers (or -1 if not allocated)
} array;
```

Before you continue, you should clearly understand where we are going. We will have a huge memory (well, not that huge) divided into frames. We will know by looking at the frame-map if a frame is taken or not. An address in an array is broken down into a page number and an offset, and a page number is translated using the page table into a frame number. Draw this on paper; once we start to manipulate these data structures, you need to be able to visualize what we're doing.

3 Creating an array

When an array is created, we need to find free frames to allocate to the array. We, therefore, start by implementing a procedure that will search through all frames and select the first free one. The frame is marked as taken, and the frame number is returned. If no free frame is found, we return -1 to signal that we have no more frames to offer.

```
int find_free() {
  for (int i = 0; i < FRAMES; i++) {
    if (framemap[i] == FREE) {
      :
    }
  }
  return -1;
}
```

We are now ready to allocate an array of a given size. We first allocate the array and the page table and then allocate the frames we need. If we fail to find a free frame, we need to return everything, and why not then use a delete procedure that we will define later? Fill in the dotted lines, and you should soon have this up and running.

```
array *allocate(int size) {
  int pages = size / SIZE;
  int rem = size % SIZE;
  if (rem > 0)
    pages += 1;

  array *new = (array*) ...
  int *pagetable = (int*) ...

  new->pages = pages;
  new->pagetable = pagetable;

  for (int i = 0; i < pages; i++)
    new->pagetable[i] = -1;    // no frame yet allocated

  printf("allocate array, frames: ");
  for (int i = 0; i < pages; i++) {
    int f = ...
    if (f == -1) {
      delete(new);
      return NULL;
    }
    printf("%d ", f);
    :
  }
  printf("\n");
  return new;
}
```

To delete an array, we simply reverse what `find_free()` does and deallocate the data structures.

```c
void delete(array *arr) {
  int pages = arr->pages;
  printf("delete array, freeing frames: ");
  for( int i = 0; i < pages; i++) {
    if( arr->pagetable[i] != -1) {
      printf(" %d", arr->pagetable[i]);
      ...  = FREE;
    }
  }
  printf("\n");
  free(...);
  free(...);
  return;
}
```

One more wrapper function that will create a new array if possible. If we fail, we have nothing else to do but fail the whole computation. No garbage collector in the world would save us since no matter how we restructure the arrays, and we will not be able to create more free frames.

```c
array *create(int size) {
  array *new = allocate(size);
  if (new == NULL) {
    printf("out of memory\n");
    exit(-1);
  }
  return new;
}
```

We are now ready to implement the set and get operations, which will be slightly more complicated than the segmentation exercise. We first need to divide the "address" into the page number and the offset. Once this is done, we check that the number is a valid page. If the page is ok, we retrieve the frame number and access the memory location.

```c
void set(array *arr, int pos, int val) {
  printf("set: arr %p  pos %d  val %d\n", arr, pos, val);
  int offset = Offset(pos);
  int page = PageNr(pos);

  if (page >= arr->pages) {
    printf("segmentation fault\n");
    exit(1);
  }

  int frame = arr->pagetable[page];
  printf("set: page %d offset %d frame %d\n", page, offset, frame);

  memory[frame*SIZE + offset] = val;
  return;
}
```

```
int get(array *arr, int pos) {
  printf("get: %p pos %d\n", arr, pos);
  int offset = Offset(pos);
  int page = PageNr(pos);

  if (page >= arr->pages) {
    printf("segmentation fault\n");
    exit(1);
  }

  int frame = arr->pagetable[page];
  printf("get: page %d offset %d frame %d\n", page, offset, frame);

  return memory[frame*SIZE + offset];
}
```

Will we catch all faulty addressing of the array? What if we allocate an array of size 40 and have a page size of 16 - what will happen when we address position 50?

4 A first run

I think you are ready to do a small test run of your system. Let's write a small benchmark to see if things work.

```
void bench() {
  array *a = create(20);
  array *b = create(40);

  set(a, 10, 110);
  set(a, 18, 118);

  set(b, 8, 208);
  set(b, 36, 212);

  printf(" a[10] + a[18] = %d\n", get(a,10) + get(a, 18));
  printf(" b[8] + b[36] = %d\n", get(b,8) + get(b, 36));

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

Also, try to create larger arrays than our memory and access outside of an array. You could try to access an array with a negative index, which will likely result in a crash. Update your implementation to catch this error and print a nice error message.

5 Be lazy

When you know that you have things up and running, it is time to do a trick that most operating systems do. We will be as lazy as possible and only allocate frames if they are actually needed. Take a look at the `allocate()` procedure; do we need to allocate all frames directly? Could we wait until we see that they are needed?

If we want to keep track of whether a frame has been allocated, we can extend the page table to hold more information. We could extend the page table entry to be a small structure that holds status information, but why not be lazy? What if we leave all entries in the page table with the value -1?

How about this:

```
array *allocate(int size) {
  int pages = size / SIZE;
  int rem = size % SIZE;
  if (rem > 0)
    pages += 1;

  array *new = (array*) ...
  int *pagetable = (int*) ...

  new->pages = pages;
  new->pagetable = pagetable;

  for (int i = 0; i < pages; i++)
    new->pagetable[i] = -1;    // no frame yet allocated

  return new;
}
```

Now we must be very careful when we write to the array. If the page table entry returns -1, we quickly need to find a free frame, insert the frame number into the page table and then continue as if nothing has happened. You can do this with just a few lines of code.

```
  :
  if (frame == -1 ) {
    printf("page fault ... ");
    frame = ...
    if (frame == -1) {
      printf("out of memory\n");
      exit(-1);
    }
    printf("ok\n");
    arr->pagetable[page] = frame;
  }
  :
```

When we fix the `get()` procedure, we realize that reading from a not yet allocated page could return zero. There is no need to allocate a frame and then do a read operation. The page has never been written to.

You are wrong if you think this is only a fun trick. This is what an operating system does every time we request more memory. It will set up the page table correctly but will not allocate any frames unless we actually write to the pages. As you will see, we can do even more tricks by delaying operations until they are needed.

6 Lazy copy

The lazy strategy can also be used when we copy an array. Why not try to delay the copying procedure until it is actually needed? The idea is to create *a lazy copy* of an array; the two array structures should share the frames. We should still be able to read from either array; only when we write to any of the two array structures will we create a copy but then only of the page that is written to.

This will require some more bookkeeping so let's extend our page table to hold something that can hold more information. Let's define a page table entry and then let the array hold a proper page table.

```
typedef enum pte_status {ALLOCATED, LAZY, SHARED} pte_status;

typedef struct pt_entry {
  int frame;
  pte_status status;
  struct pt_entry *copy; // who else shares the frame
} pt_entry;

typedef struct array {
  int pages;
  pt_entry *pagetable;
} array;
```

This is much more interesting. An entry could now be either properly *allocated*, a *lazy* allocation that we should fix (the -1 that we used before), or a *shared* frame. If it is a shared frame, we also have a pointer to the page table entry that is the lazy copy. This will be a bit tricky so buckle up.

First, you should go through your code (or why not create a copy and work on the copy) and update the code so that it works with the new representation of page table entries. Before treating it as a frame number or -1, we must look inside the data structure to figure out what to do. If you look at this updated version of `allocate()` you will be able to update also `delete()`, `set()` and `get()`.

```
 :
array *new = (array*)malloc(sizeof(array));
pt_entry *pagetable = (pt_entry*)malloc(sizeof(pt_entry)*pages);

new->pages = pages;
new->pagetable = pagetable;

for (int i = 0; i < pages; i++) {
  pt_entry *entry = &new->pagetable[i];
  entry->copy = entry; // a trick
  entry->status = LAZY;
}
  :
```

If you can run your previous benchmarks, you should be fine. Now for the tricky part, how do we implement `copy()` and what changed do we have to do to the implementation of `set()`, `get()` and `delete()`? The idea is like this, a page table entry could be in either of three states:

- ALLOCATED: a frame has been allocated for the page, and the array is the only user of the frame.
- LAZY: a frame has not yet been allocated but will be as soon as a set or get procedure is called.
- SHARED: a frame has been allocated, but the page is shared by two or more arrays. As long as we read from the page, no harm is done, but as soon as we do a write operation, we need to create a copy.

We write "two or more" since we need to handle the case when we have taken a copy of a copy. This will complicate things since we need to keep track of which other arrays share the page, and when there is only one left, it should treat it as its own allocated frame.

To keep track of who else shares the frame, we link all page table entries that share a frame in a circular list. If an entry is the only entry that holds a reference to the frame, i.e., ALLOCATED, the copy reference is a circular pointer to the entry itself (this is a trick that will make our coding easier).

This sounds complicated, and it is. Make some drawings of what things could look like. Also, write down what should be done when we create a copy of a page table entry in your own words.

- ALLOCATED: ... now shared ... SHARED
- LAZY: this is easy .... LAZY
- SHARED: ... could also share ... linked in a circular ...

When you have done some drawings, you are ready to copy an array; this skeleton code should give you a head start.

```
array *copy(array *orig) {
  array *copy = (array*)malloc(sizeof(array));
  pt_entry *pagetable = (pt_entry*)malloc(sizeof(pt_entry) * orig->pages);

  int pages = orig->pages;
  copy->pages = pages;
  copy->pagetable = pagetable;

  for (int i = 0; i < pages; i++) {
    pt_entry *orig_entry = &orig->pagetable[i];
    pt_entry *copy_entry = &copy->pagetable[i];

    switch (orig_entry->status) {
      case LAZY:
        copy_entry->status = ...
        copy_entry->copy =  ...  // circular
        break;
      case ALLOCATED:
      case SHARED:
        copy_entry->frame = ...
        orig_entry->status = ...
        copy_entry->status = ...
        // linking in the circular structure
        copy_entry->copy = ...
        orig_entry->copy = ...
        break;
    }
  }
  return copy;
}
```

Notice that we do the same if the original entry is allocated or shared. Suppose the original entry has an allocated frame. In that case, we simply mark this as a copy and create the initial circular structure knowing that an allocated array has a self-reference in the *copy field*. That was not that complicated because the complicated copying is delayed until we do a `set()` operation. When we do a set operation, we need to take care of the situation when two or more arrays share the same page; if an entry is referring to an allocated page or a lazy page, the situation is as before. This is what we need to insert:

```
:
if (entry->status == SHARED) {
  int f = find_free();
  if (f == -1) {
    delete(arr);
    exit(-1);
  }

  for (int i = 0; i < SIZE; i++)
    memory[f*SIZE + i] = memory[entry->copy->frame*SIZE + i];

  entry->frame = ...
  entry->status = ...

  // find the entry that is previous to entry
  pt_entry *prev = entry;
  while (prev->copy != entry)
    prev = prev->copy;

  // it should now point to ...
  prev->copy = ...

  // and if it is pointing to itself ...
  if(prev->copy == prev)
    prev->status = ...
}
:
```

Ahh, not that complicated (it took me an hour) after all? The `get()` procedure does not need any changes since reading from an allocated or shared frame will be the same. You might want to write this as a switch statement to make it obvious, but nothing special needs to be done. The `delete()` procedure needs to be updated since we could now delete an array that holds a shared frame. The procedure is then similar to the last part in the set operation, i.e., we unlink the entry from the list of copies, and if there is only one copy left, that entry is changed to allocated.

The lazy copying you have now implemented is what the operating system does every time you do a `fork()`. The code area is read-only and will be shared until either process calls `exec()`. The data areas will be shared until written to, but only the pages that are written to will be copied.