CS 341 – Assignment #3
Problems 1, 2 and 3: due 11-May-2011
Problem 4: due 25-May-2011

1 Problem 1

3.57 from the book.

2 Problem 2

Reoragnize the following C structure to have optimal packing.

```
struct S {
  char c1;
  int *p1;
  unsigned short s1;
  uint64_t ull1;
  char c2;
  double d1;
};
```

Provide:

- The size, in bytes, of the original ordering on x86.
- Justification for the ordering that you choose.
- The size, in bytes, of the optimized ordering on x86.

3 Problem 3

Implement a binary search routine over integers in assembly. The function should be C callable with the following prototype:

```
/**
 * Search an array for a key value
 *
 * \param data  Pointer to the data to be searched.
 * \param n     Number of data elements stored in data.
 * \param key   Value to find.
 *
 * \return
 * The index in \c data containing \c key if found, or -1 otherwise.
 */
int search(const int *data, int n, int key);
```

Use the code in the accompanying `unit_test.c` to verify your implementation. Add the following to the top of your `main.cpp`

```
extern "C" int check(void);
```

and call `check` from your `main`.

4 Problem 4

Implement a n-bit unsigned integer radix sort in x86 assembly. Assume that n is a compile-time constant of your choosing. Either 4 or 8 is a good value. The function should be C callable with the following prototype:

```
/**
 * Sort the data in increasing (lowest to highest) order
 *
 * \param data    Pointer to the data to be sorted.
 * \param count   Number of data elements to be sorted.
 */
void sort(unsigned *data, unsigned count);
```

Use the code in the accompanying `sort_test.c` to verify your implementation. If you get compilation errors in `sort_test.c`, try commenting out the `#define DO_TIMING` line. Add the following to the top of your `main.cpp`

```
extern "C" int check(void);
```

and call `check` from your `main`.

Your initial implementation should use a large temproary data buffer as temporary storage for the par-sorted array. Do this by creating an uninitialized data section (also called BSS). Code NASM code below creates a variable called `temp_space` that is, essntially, an array of (512×16) 32-bit words (either `int` or `unsigned` storage). The code from the previous assignment already had an empty BSS section. You can simply add the `temp_space` declaration to that existing section.

```
        [section        .bss align=16]
temp_space      resd    512 * 16 ; reserve space for 512 * 16 unsigned integers
```

Your final implementation should call `malloc` and `free` to alocate temporary storage. You will need to make several additions to your assembly code to do this.[1] These are:

- Add external declarations for the functions that you will call:

```
extern    _malloc, _free
```

- Add the call to `malloc`:

```
push    eax                 ; amount of space to allocate
call    _malloc
add     esp, 4              ; keep the stack balanced!
mov     [ebp - 8], eax      ; store the returned pointer somewhere
```

- Add the call to `free`:

```
push    dword [ebp - 8]     ; pointer to the data to be freed
call    _free
add     esp, 4              ; keep the stack balanced!
```

Remember that after calling either `malloc` or `free` the values previously in `EAX`, `ECX`, and `EDX` will be lost.[2] The pointer returned by `malloc` can either be stored in local storage on the stack frame or in a register. The offset from `EBP` to the local stack frame is just an example.

In either case, only one block of storage needs to be allocated. This block must be partitioned into per-bucket blocks. The easy way to do this is to allocate enough storage so that each bucket has space for the entire array. The entire storage space is then partitioned into fixed size chunks. In C, this would look like:

[1] See `http://www.caswenson.com/past/2009/9/26/assembly_language_programming_under_os_x_with_nasm/` for more details.
[2] See `http://www.agner.org/optimize/calling_conventions.pdf` for more details.

```
unsigned *buckets[NUM_BUCKETS];
unsigned *storage = malloc(sizeof(unsigned) * n * NUM_BUCKETS);

for (unsigned i = 0; i < NUM_BUCKETS; i++)
        buckets[i] = &storage[i * n];
```

On each pass through the main loop, data is copied into the proper buckets. When this kind of partitioning is used, at the end of the data is copied out of the buckets into the original storage.

At least two pieces of stack storage are needed to implement the radix sort. Storage is needed for the bucket pointers (called `buckets` in the C code above), and storage is needed for the counters of the number of elements in each bucket. Like most assemberls, NASM is a *macro assembler*. This means it has some features somewhat like the C preprocessor to make programming a little less painful. In NASM you can create defines to make keeping track of magic values in the code a little easier. The NASM documentation[3] explains all of these features. For example, you might do the following:

```
%define RADIX_BITS              4
%define RADIX_BUCKETS           (1<<4)

        ; Offsets to parameters passed in
%define DATA_OFFSET             8
%define COUNT_OFFSET            12

        ; Offsets to storage for non-scratch registers.  These are examples.
%define ESI_SAVE_OFFSET         (-4)
%define EBX_SAVE_OFFSET         (-8)

        ; There is one bucket pointer (4 bytes) for each bucket
%define BUCKET_POINTER_OFFSET   (-(RADIX_BUCKETS * 4) + ECX_SAVE_OFFSET)
        ; There is one bucket counter (4 bytes) for each bucket
%define BUCKET_COUNTER_OFFSET   (-(RADIX_BUCKETS * 4) + BUCKET_POINTER_OFFSET)

        [global _sort]
_sort:
        push    ebp
        mov     ebp, esp
        add     esp, BUCKET_COUNTER_OFFSET      ; Reserve space for locals
        mov     [ebp + ESI_SAVE_OFFSET], esi    ; Save esi
        mov     [ebp + EBX_SAVE_OFFSET], ebx    ; Save ebx

        ; code goes here

        mov     esi, [ebp + ESI_SAVE_OFFSET]    ; Restore esi
        mov     ebx, [ebp + EBX_SAVE_OFFSET]    ; Restore ebx
        mov     esp, ebp
        pop     ebp
        ret
```

By doing this, the code is a lot more readable (e.g., it's more obvious that you typed `EBX_SAVE_OFFSET` vs. `ESI_SAVE_OFFSET` than -8 vs. -4). The code is also a lot easier to change. If you decided to add another stack variable or another "save" location, only the defines need to be updated (as opposed to updating every place the offsets are used).

Extra credit: Allocating $O(2^n m)$ memory, where n is the number of radix bits and m is the number of elements in the array to be sorted, for temporary storage is wasteful. It is possible to implement a radix sort using only

[3] `http://www.nasm.us/xdoc/2.09.08/html/nasmdoc4.html`

$O(m)$ memory for temporary storage. This requires doing extra work to calculate how many slots are needed for each bucket. There are several benefits to this approach:

- Use less memory.
- Avoid the extra copy at the end of the main loop.
- Eliminate the need for the bucket usage counters. These are only needed in the basic implementation because the bucket base pointers are needed for the extra data copies. Note, however, that usage counters for the *next* radix implementation are still needed to partition the bucket storage space on the next pass.

Extra credit will be given for implementations that use only $O(m)$ memory for temporary storage and avoid the extra copy. I *strongly* recommend doing the simpler implementation first. This implementation is a bit more complex, and you're much more likely to be successful starting from a working sort routine.