UNIVERSITY of York

SCHOOL OF PHYSICS, ENGINEERING AND TECHNOLOGY

Systems Programming for ARM Laboratory Scripts

Dr. Andy Pomfret

Laboratory 5

Memory Allocation

5.1 Aims

In this laboratory session, you will:

- Learn why dynamic memory allocation is discouraged in embedded systems;
- Create and use a flexible memory pool;
- Understand the reasons for choosing static memory allocation, and investigate pseudo-static allocation at runtime.

ⓘ You have probably encountered the standard library functions `malloc()` and `free()`. `malloc()` takes a number of bytes as its only parameter, and returns a pointer to a block of memory of at least that size that is not being used for anything else, in the form of a `void *`. It will return a null pointer (zero) if the allocation fails. The memory comes from an area known as the "heap", though `malloc()` does not necessarily actually use a heap structure for the storage of available memory blocks. `free()` takes a pointer to a previously allocated block and returns it to the heap.

However, managing memory using `malloc()` and `free()` has a number of disadvantages:

- *Speed.* Allocating memory from the heap is time-consuming, because the heap must be searched until a large enough free block is found. Deallocation is faster, but most modern implementations of `free()` try to merge adjacent free blocks to minimise fragmentation of the available memory, and this takes time.

- *Size.* For the same reason that the routines are slow – their complexity – they are also large. Using `malloc()` and `free()` causes them to be linked into your compiled code, and this can increase the size to an undesirable extent.
- *Safety.* It's easy to forget to free allocated blocks of memory, which leads to a situation called a *memory leak*. It's also remarkably easy to free the same block twice, which leads to undefined behaviour, or to pass an invalid pointer to `free()` (something that's not been allocated with `malloc()`), and again undefined behaviour is the result.
- *Utility.* On a desktop computer, where it is not known in advance what software will be running, flexible dynamic memory allocation is essential. But on an embedded system, it's difficult to see what it adds: if you can't place an upper bound on the amount of memory a process will need, or if you can but that bound is higher than the amount of available RAM, then you have a problem that goes way beyond memory management!

⚠ For these reasons, the use of dynamic memory allocation using `malloc()` and `free()` is, in general, considered **bad practice** when developing for embedded systems. Many companies and organisations have an outright ban on the use of these functions in embedded code. When using the full standard C library, as we are doing, it is not possible to completely disable the system heap because it is used internally. Smaller embedded implementations of most of the C library, like picolibc, will permit this. But either way, you should **avoid** using these functions in your own code.

Being able to allocate blocks of memory for a specific purpose at runtime is still potentially useful, however. In this lab we will look at two tools that can be used to address this: *pseudo-static allocation* for memory that will only be allocated once and never freed, but where the required amount of memory may not be known until runtime or where calculating it at runtime may make your code neater; and *memory pools*, for memory that can be dynamically allocated and then released when no longer needed.

But first, we'll briefly recap the idea of static storage and how it can be helpful in an embedded systems context.

5.2 Static Variables

We have previously looked at the effect of the **`static`** keyword in terms of the behaviour of variables (lifetime and visibility). But statically-allocated variables have another very important use: reducing stack usage.

Recall that `auto`-class local variables are allocated space on the stack. This can, especially if the variables in question are large, lead to the phenomenon of *stack overflow*, where new local variables are allocated to space beyond the end of the stack, and end up modifying memory that's being used for something else. This is (needless to say) Bad, and is also often very difficult to diagnose — the symptoms are very weird, as things unrelated to the actual problem appear to break at random.

Of course, it is possible to increase the amount of stack space available to mitigate the problem of stack overflow. But this is not very efficient; the allocated stack must be able to cope with the *worst case* stack usage, but the *average* stack usage is likely to be very much smaller. On a simple single-tasking system that's not really a problem — or rather, if there's not enough RAM for all the possible local variables, you already have a bigger problem than how your variables are allocated. But if you're using an operating system where each running task has its own stack, which is typical, then it can make a huge difference to the total amount of required RAM.

ⓘ There are three alternatives to `auto`-class local variables:

- `extern`-class global variables. Using these as a replacement for locals is a **bad idea** because they are visible to the linker.
- `static` global variables. These can be used, but have the disadvantage of having a much broader *scope* (that is, they can be seen and possibly accidentally modified by other functions in the same file).
- `static` local variables. These can have the same scope as the `auto`-class variables they replace, but some programmers dislike them because their behaviour is different from ordinary local variables due to the different lifetime.

My preference is for `static` locals. Some employers will have coding guidelines that prefer either these or `static` globals.

⚠ Whether `static` locals or `static` globals are chosen, there is a downside: code that uses these in place of `auto`-class locals will not be *reentrant*. Reentrancy is the property that allows a function to be called more than once *at the same time*. This covers recursion, where a function calls itself, as well as concurrent access when an operating system is being used. If `auto`-class variables are used then each running instance of the function has its own set of variables; if not, then the variables are shared between instances and the function stops being reentrant.

ⓘ **TL;DR**: If you're writing a function that doesn't need to be reentrant, you should prefer `static` variables for any local variables over a few bytes in size. Simple integer and floating-point types can still be `auto`-class.

5.3 Pseudo-Static Allocation

Sometimes it can be useful to define the size of some required memory in terms of something that is only known (or only easily computed) at runtime. For example, it might be useful to declare an array, statically, with a length equal to the number of tasks that have been declared in the context of an operating system (information which is known at compile time, but which may be difficult to compute without using code) or given by the user during a setup process, for example.

Since C99, the C language has had support for "variable length arrays", which are not actually *variable* length, but have runtime-determined lengths. However these are only available for `auto`-class locals. Any variable declared as `static` must have a size that's known at compile time. So how can arrays like this be allocated statically?

Use Project→Open Project... to open `Allocation\static\static.uvprojx`.

The code in `main()` creates and fills two arrays based on the value of two variables. This is a contrived situation of course, and compiler optimisation might even identify the constant nature of the variables in question, but it serves to show the issue.

Build the code. Note that the array declarations generate warnings: this is precisely because the amount of stack space that variable length arrays will use is, in general, indeterminate. This compiler warns about this by default. Most others do not.

Try declaring one or both of the arrays to be `static`. What happens?

5.3.1 Completing the Allocator

The files `static_alloc.c` and `static_alloc.h` contain the framework for a simple runtime static allocator.

Take a look in `static_alloc.c`. Notice:

- The size of the static pool is defined using a preprocessor symbol
- A static array is declared, containing as many bytes as required
- An index into the pool array is declared, and initialised to the size of the array
- The `static_alloc()` function is declared, returning a `void *` and accepting a single parameter indicating the required allocation size.

The `size_t` type exists to provide a platform-independent way of describing the sizes of things, in bytes. It is an unsigned type; its size is implementation defined, but it will always be capable of describing the largest possible logical object on the target platform. It is the type returned by the `sizeof` operator, and should always be the type used for this purpose. It is defined in `stddef.h`, which is included by many other system headers.

The index is initialised to the size of the pool array, rather than to zero, because it is indended that memory will be allocated from the array "backwards", with the index decreasing on each allocation. This is for two reasons, both of which will make your code a little simpler.

Complete the `static_alloc()` function. It should:

- Check that the requested number of bytes is available in the array. This is easy because the pool array index decreases: if the index is greater than the number of requested bytes, all is fine. If not, zero should be returned to indicate failure;

- Reduce the pool array index by the requested number of bytes;
- Return the new pool array index.

Modify the code in `main()` to use the new function:

- Include the `static_alloc.h` header in `main.c`;
- Replace the array declarations with calls to `static_alloc()`, so e.g.

  ```
  uint32_t a[len_a];
  ```

  should become

  ```
  uint32_t * a = static_alloc(len_a * sizeof(uint32_t));
  ```

Build and run your code. Does it work as expect? If you use the debugger and step into the `static_alloc()` function, does the index decrease as you expect?

Strict Aliasing

This static allocator takes a pointer to a `uint8_t` and returns it as a `void *`. The code that has called the allocator then converts this to a pointer to something else, for its own use. Is this a violation of the strict aliasing rule?

The answer is... *maybe*. The C standard is not sufficiently clearly written to know for sure. At no point is this program writing data into a memory location and then reading it as if it was a different type, which is arguably the intent of the strict aliasing rule, but that's not actually how it's written. The "object" to which the returned pointer refers is a `uint8_t` (or rather, four of them), but the returned pointer is treated as referring to a `uint32_t`.

⚠ At the function level, there is no violation; at the translation unit level, there is no violation; but at the program level, there *could* be. There is no clear consensus about whether or not this code is permissible according to the standard. If it is not, however, then it is arguably impossible to write *any* kind of allocator that complies with the strict aliasing rule.

The traditional C build model, where compilation and optimisation is applied at the level of the translation unit, will ensure that no problems occur as a result of this possible undefined behaviour. But as compilers become more sophisticated and increasingly apply optimisations that cross translation unit boundaries, that is no longer guaranteed. The best hope for continued behaviour of code like this is that it is very common, and widely supported by compilers. Perhaps future versions of the C standard will clarify this situation. (C11 and C17 did not.)

Perhaps the lesson here is that although it is very important to understand the strict aliasing rule, and it is vital to adhere to it at the translation unit level, systems programming will always involve working on the murky edges of the language.

5.3.2 Alignment

The existing implementation has a flaw: the returned memory blocks are potentially *unaligned*. Remember, alignment of data items to particular boundaries can be important for performance, and as we'll see it's sometimes essential for complying with various interoperability requirements.

It is conceptually reasonably simple to arrange for the static allocator to return blocks aligned to arbitrary boundaries. Because the index decreases with each allocation, it is sufficient simply to round the index down to the next alignment boundary when computing it and before returning it. Rounding down to the nearest alignment boundary is also quite simple because alignment boundaries are always powers of 2, so this can be achieved simply by setting the least significant bits of the index to zero. In turn, this can be a done using a simple bitwise "and" operation.

Modify your code to allocate aligned blocks. My advice would be to:

- Define new preprocessor constant `STATIC_ALLOC_ALIGNMENT` and set it equal to 8, for double-word alignment.
- Work out a way of converting the alignment value to a *bit mask*, which has every bit set to 1 except a few of the least-significant bits. It's ok to assume that the alignment value is a power of two. Hint: the binary representation of $2^n - 1$ has ones in bits 0 to $n - 1$, and zeros elsewhere.
- Use the bitwise "and" operator `&` to round the index down to the next alignment boundary when computing it.

This code ensures that the *index* is aligned, but that will only translate to the returned *pointer* being aligned if the pool array is also aligned. How can we ensure this?

There is no way using plain C to do this, except by allocating a block that is too large and then rounding up its base pointer to the next multiple of the alignment size. However, most compilers will allow alignment to be specified on data, and the most popular compiler toolchains (GCC and clang) share a syntax for this. It's not a standard, but it's reasonably portable.

Change the declaration of the pool array to read (all one line)

```
static uint8_t static_pool[STATIC_ALLOC_POOLSIZE] __attribute__ (( aligned(
   STATIC_ALLOC_ALIGNMENT) ));
```

Note that the double set of parentheses is **required**. The syntax for all GCC/clang attribute decorations is

```
__attribute__ (( ... ))
```

Build and test your code. Perhaps use `static_alloc()` to allocate some blocks that are not a multiple of 8 bytes in size, and check that all blocks are returned aligned (and do not overlap!).

5.4 A Memory Pool

Memory pools provide an alternative to `malloc()` and `free()` that is faster and smaller. (The safety issue is not resolved, however.) The efficiency savings come because the blocks of memory handled by a pool are *all the same size as each other*. If a program requires blocks of different sizes for different purposes, multiple pools must be used.

The simplest implementation of a memory pool uses a singly-linked list. Unused blocks of memory are stored in a linked list such that allocation and deallocation interacts only with the head of the list: allocation returns the first item in the list and updates the head pointer, and deallocation inserts a new item at the head of the list and links it to what was previously the first item. The available data blocks must all be added to the pool when the program begins to run.

The interesting part is that the list pointers *overwrite* part of each data block. This doesn't matter because, by definition, the blocks contain no important data when they're in the pool; and when they're removed from the pool the list pointer isn't important and can be overwritten.

Use Project→Open Project... to open `Allocation\mempool\mempool.uvprojx`.

This program uses a memory pool to allocate space for some structures, which it populates with data. It prints the data from each structure, along with the memory address where the structure is located, to show that the pool is working. However, the implementation is not complete.

There are two functions in `mempool.c` – `pool_allocate()` and `pool_deallocate()` – that are not complete and need writing. Note that the other function used by `main()`, `pool_add()`, is simply aliased to `pool_deallocate()` by a **`#define`** directive in `mempool.h`. (If you're not clear why deallocation is the same operation as initialisation, please ask a demonstrator.)

Your task is to write the two missing functions.

- `pool_deallocate()` takes two arguments, a pointer to a `pool_t` structure and a pointer to a block of memory that is to be added to the pool, either because the pool is being initialised or because the block was previously allocated from the pool and is no longer needed. It must insert a pointer into this block that points to the current head of the list, and then update the head pointer to point to the new block.
- `pool_allocate()` takes a pointer to a `pool_t` structure as its only argument. It must return a pointer to an available block by returning the head pointer, and it must also update the head pointer to point to the next available block. If the pool is empty, it must return the null pointer (`0` or `NULL`).

Test your code by checking that the remainder of the provided code works as intended.

5.4.1 Improving the Interface

Having the user add blocks to a pool manually is a little cumbersome. Wouldn't it be good to be able to declare a pool and have it ready for use automatically?

Unfortunately this isn't possible because initialisation of the pool requires looping to link the blocks together, and C doesn't support the concept of static code in the sense of Java's static blocks, and neither does it allow code to be executed when a variable is defined (there's no such thing as a "constructor" in C).

However, we can do the next best thing: create a function that initialises and populates a memory pool all in one go, without requiring the user to create their own array and loop. A function that did this would need to statically allocate enough memory for the items in the pool, but you have already written tools that will allow that to happen!

Using the Windows file explorer, find the completed static allocation files `static_alloc.h` and `static_alloc.c` from earlier on and copy them to the `inc` and `src` folders respectively in the project you're working on. Now right-click the "Source Files" group in the project pane on the left of the µVision window and click "Add Existing Files to Group 'Source Files' ". Navigate to the `src` folder of the current project (take care with this, the browser often defaults to a location inside a *different* project!) and add the `static_alloc.c` file. This will cause it to be compiled when the project is built.

Now try writing a function that will allocate storage and use that to fill a memory pool. The function signature should be

```
void pool_init(mempool_t *pool, size_t blocksize, size_t blocks);
```

Remember to add the prototype into `mempool.h` as well as creating the implementation in `mempool.c`. You may need to include `stddef.h` to get access to the `size_t` type.

The function should:

- Allocate enough storage to hold all of the requested blocks, using `static_alloc();`
- Loop once for each requested block, compute the address of the start of the block, and pass it to `pool_add()`.

Remember to check the return value of `static_alloc()` to ensure that the allocation succeeded. If it didn't, the most sensible course of action would probably be to set the pool head pointer to zero so that it's incapable of allocating blocks.

You may wish to round up the block size to the nearest multiple of `STATIC_ALLOC_POOLSIZE` before you start, so that all blocks in the pool have the same alignment as things that are allocated by `static_alloc()`. You'll need to move this to `static_alloc.h` to gain access to it. Perhaps now its name is inadequate given that it describes allocation of memory pool blocks as well as pseudo-static allocations. Maybe it would be sensible to combine all of this allocation code into one source file and one header file...

5.5 Summary and Conclusions

By now, you should:

- Know why avoiding large `auto`-class local variables is desirable in an embedded systems context;
- Understand why, in the same context, dynamic memory allocation is undesirable;
- Know how memory pools and pseudo-static allocation can be used to reduce stack usage without recourse to the heap.

If you have any questions about what you have learned, please ask.

Remember to copy your project files to a network drive before you log off.