University College London,
Department of Computer Science

An investigation into the fragmentation performance of the Symbian Operating System memory allocator, and a critical assessment of the extensibility of the allocator framework

Author: Dan Kolb
Supervisor: Chris Clack

Submission date: 7th September 2004

Abstract

Name:	Dan Kolb
Supervisor:	Chris Clack
Project Title:	An investigation into the fragmentation performance of the Symbian Operating System memory allocator, and a critical assessment of the extensibility of the allocator framework

This report initially presents an investigation into the performance of the memory allocator on the Symbian Operating System, which is a first-fit sequential fit allocator that uses coalescing. Traces are obtained from various smartphone applications, and are presented in a style that is consistent with **Johnstone and Wilson**[8].

Profiles of the distribution of allocated objects in each application are also presented in a style that is consistent with an investigation by **Bohra and Gabber**[1] into memory fragmentation issues.

Finally an assessment of the extensibility of the Symbian OS allocator is given. The new Operating System that is currently in development contains facilities to implement a new memory allocator, overriding their implementation – this framework is given a critical assessment with an attempt to implement a segregated free list allocator.

Contents

Chapter 1

Introduction

This section gives an overview of memory management and fragmentation issues arising from dynamic allocation of memory, and why memory management on low-memory devices such as smartphones is important. Aims and objectives of the project will also be described, and a brief overview of the structure of the report will be given.

1.1 Setting the scene

Memory management is often thought of as a solved problem – with exponential growth of memory sizes in the last few years, attempts at optimising to save a few kilobytes (or even a couple of megabytes) seem futile; in most interactive devices memory can easily be upgraded if necessary. However, memory is still a limited resource, and therefore requires some careful conservation and recycling. This recycling of unused memory in an area known as the 'heap' forms the basis of dynamic memory management.

In C and C++, the programmer must specify when to allocate and free dynamic memory, while in other languages such as Java, the `new` operator allocates memory for an object, and when the object is not in use any more, it gets garbage collected – the garbage collector automatically reclaims memory that is deemed to be unused without the programmer needing to explicitly free the the memory. An indication as to how efficiently this heap is being managed is given by the amount of memory fragmentation that occurs.

The phenomenon of memory fragmentation occurs in two forms, external and interal. External fragmentation arises when a memory request from the program cannot be fulfilled, even though sufficient free memory exits. This is due to the free memory being scattered over the heap in small blocks, rather than in one contigous block (which is what is needed to satisfy a memory

request). Conversely, internal fragmentation occurs in the situation where the program is returned a block of memory that is greater than that specified by the request. This results in a portion of the allocated memory being left unsued, which forms the basis for internal fragmentation. Both forms of fragmentation arise through (repeated) inefficient memory allocation and deallocation. To some extent, fragmentation is unavoidable in a dynamic memory system; however, it can be kept to a minimum by using a good memory allocator.

One of the results of fragmentation is that more memory is used than is necessary, which is wasteful of a limited resource. In extreme circumstances, a program may terminate because all the memory is used up by inefficiencies in the memory allocator, even though the program itself is only actually using a small amount of memory. It was this issue that prompted a study by Bohra and Gabber [1] into memory fragmentation issues.

There are a number of different allocation policies that can be used, which are described in section 2.2. Some of these give extremely fast allocation at the expense of poor storage utilisation (high fragmentation), whereas others are optimised for storage utilisation at the expense of having slower allocation times.

1.1.1 Low memory devices

An area where efficient use of memory is essential is in low-memory devices, such as smartphones. These tend to have between 8MB and 64MB of RAM, and hence using memory efficiently is crucial to the performance and functionality of these devices; it would be unacceptable to have, for example, a heap size that is 4 or 5 times the size of the maximal live memory usage (as has been found in some allocators tested in [8]). Additionally, due to slower processors in these devices, memory allocation and deallocation ought to be as fast as possible to give a more responsive feel to the applications – i.e. the OS does not spend a long time allocating memory (this, however, is becoming less of an issue as smartphone memory access speeds increase, so the CPU is able to access memory faster and is therefore not kept waiting).

1.2 Aims and objectives

This report has two main aims – one is an investigation into the fragmentation performance of the Symbian memory allocator, and the other is an assessment of the extensibility of the allocator framework.

The investigation into the fragmentation performance involved obtaining

traces of the allocation, free and reallocation operations from running various smartphone applications under the Symbian OS. This appears to be the first study to obtain memory traces from smartphone applications – other studies have typically focused on applications running on workstations or servers. Results of the performance are to be presented in a style similar to [8]; this will allow for a comparison between the fragmentation of the Symbian allocator and fragmentation results obtained by [8] for a similar allocator. A subsidiary aim of the tracing is to be able to map out the frequency of object sizes (i.e. how many times an object of a particular size is allocated); this is presented in a style similar to some characterisation of applications found in [1].

The assessment of the extensibility of the allocator involved attempting to implement a new allocator and use it within the extensibility framework provided by the Symbian allocator. This is intended to provide a starting base for anyone interested in developing their own memory manager in place of the existing Symbian OS manager, and also to provide Symbian with an assessment of their own framework from the point of view of an external developer, thus giving an indication of how the framework may be improved.

1.3 Report structure

Chapter 2 gives an introduction to memory allocation from a programmer's perspective, and follows with a description of some of the most common memory allocation policies currently in use. An introduction to the Symbian Operating System as well as coding conventions and compilation methods is given – this is most useful in assisting with understanding some of the code given in chapters 4 and 6. Finally two papers, by Johnstone and Wilson [8] and Bohra and Gabber [1] are summarised. These papers were central to the investigation, being used as a reference for the assessment of the Symbian memory allocator, and the characterisation of the applications.

A review of the Symbian OS framework is given in chapter 3. This covers the operation of executable files (and libraries) in the Symbian OS, provides a description of their implementation of a memory manager, and goes on to describe the framework in place to be able to extend the memory manager.

Chapter 4 describes the design and implementation of a set of tracing extensions to the Symbian allocator to generate performance statistics, covering the method of processing the traces and any issues that occurred during the implementation. The results from the traces are presented in chapter 5.

A critical assessment of the Symbian OS framework for extensibility by attempting to implement a segregated free list allocator is presented in chapter

6. This includes design considerations and a description of the implementation of the allocator, and also describes any issues that occurred.

A summary of the work performed is given in chapter 7. This is followed by suggestions for further work, and a conclusion, which includes some personal comments about the work.

Chapter 2

Background

This chapter gives an introduction into how memory allocation and deallocation works from a user perspective (the functions used to allocate and deallocate memory) and goes on to describe various common algorithms used in the allocation process. An introduction to the Symbian OS and smartphones is given, including topics such as naming conventions, and an introduction to program compilation (both of which will assist in understanding some of the implementation, given in chapter 4). Summaries of the **Johnstone and Wilson**[8] and **Bohra and Gabber**[1] papers are also given. The **Johnstone and Wilson** paper is widely regarded as a canonical investigation into the fragmentation performance of various memory allocators when running a number of 'real and varied' programs; Bohra and Gabber continue to investigate memory fragmentation using two more specialised programs, as well as providing some characteristics of the objects allocated by the applications themselves.

Sections 2.1 here to 2.1.2 are taken and paraphrased from [9].

2.1 Memory allocation procedures

Most programs need to dynamically allocate and deallocate storage during the course of their execution, in addition to the static storage reserved by the compiler at run-time. This is normally implemented with a mixture of using a stack and a heap.

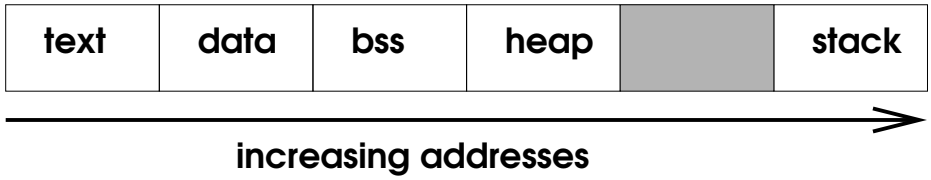

Figure 2.1: Schematic map of a process address space

2.1.1 Stack

A stack is usually placed at the top of the address space, from where it grows down (see figure 2.1). There tends to not be any kernel interface to the stack – growing and shrinking is handled automatically as the stack is accessed. If the stack fails to be extended for some reason, then the process requesting stack space is normally terminated [9]. However, what often occurs (unless the stack has its own memory segment) is that stack overrun happens long before the stack fails to be extended. Overrun often causes data to be corrupted, and, eventually, the executable code in memory is also corrupted.

The C programming language does have an interface to explicitly allocate space on the stack, `alloca(3)`, but its use is discouraged[1], as the function is machine and compiler dependent, and on many systems the implementation is buggy[2]. The function is not an interface to the kernel, but an adjustment done to the stack pointer such that space will be available and unharmed by any further subroutine calls. When the current function returns, the space is automatically returned when the stack frame is dismantled. This asymmetry can cause problems, and is one of the reasons that `alloca(3)` is not widely used[9].

2.1.2 Heap

The heap extends the data segment of the process, starting at the end of the *bss*[3] section[4], and extending upwards. The storage is explicitly allocated with the `brk(2)` and `sbrk(2)` system calls under Unix; under the Symbian OS, a similar system call, named `SetBrk` is used. The `SetBrk` and `brk(2)` functions take one argument, which is a pointer to where the process wants

[1] the function is not in the POSIX specification

[2] read the *BUGS* section in the manual page for `alloca(3)`

[3] block started by symbol

[4] this section contains uninitialised variables, and stems from a PDP/11 assembler mnemonic

the heap to end[9]. These are not particularly convenient interfaces to deal with – it is easy enough to allocate memory with it, but freeing up blocks can only occur in a LIFO order. For this reason, the `malloc(3)`, `realloc(3)`, and `free(3)` interfaces (or their equivalents in other programming languages and operating systems) are preferred for use, unless the program in question implements its own memory management facilities. The particular functions used in the Symbian OS are given in section 2.3.1.

2.1.3 `malloc`

The `malloc` function allocates a block of memory of a requested size. The C definition (as defined in `stdlib.h`), is:

```
void * malloc(size_t size);
```

This will allocate a block of at least size `size` bytes, if it can. If the call is successful, it will return a pointer to the start of the data area. If not, it will attempt to ask for more memory through the kernel, then allocate that; if that fails it will return `NULL`, which has to be explicitly checked for in the calling procedure. C and C++ do not have any built-in bounds checking, so if any memory is read from, or written to, past the end of the requested area, unpredictable results may occus, including the program terminating with a Segmentation Fault (core dumped).

2.1.4 `free`

The `free` function frees up allocated memory. The C definition for this function is:

```
void free(void *ptr);
```

This will mark a block as being free, thus enabling its reuse. The pointer `ptr` needs to be an address that has previously been returned by a call to `malloc` (and not previously freed), otherwise problems may occur (behaviour in this case is undefined).

2.1.5 `realloc`

This function reallocates memory, and normally makes an attempt to not move the block. The definition of this function is:

```
void * realloc(void *ptr, size_t newsize);
```

If the `ptr` argument is `NULL`, then the function works like `malloc(3)`. If the requested size, `newsize` bytes, is smaller than the current length of the block, then the block is shrunk, with a new block being (if possible) created from the memory size difference between the old and new sizes and added to the list of free blocks, and hence the pointer is unchanged. If the requested size is larger, then (usually) an attempt is made to grow the block in-place. If this is not possible, then a new section of memory is allocated, the data copied across, and the current block is freed up; a pointer to the new data area is then returned. If, however, the extra allocation fails, then the pointer returned is `NULL`, but the old data area remains intact. It is therefore necessary to be careful to not assume that an extending `realloc` operation will succeed by immediately by using code such as (taken from the manual page for `realloc(3)` in OpenBSD):

```
size += 50;
if ((p = realloc(p, size)) == NULL)
    return NULL;
```

This would cause a memory leak as the pointer to the data area that `p` pointed to has been lost, and hence that memory can never be freed up. Additionally, aberrant behaviour may occur if `size` is being used to track the total memory used.

2.2 Allocator Implementations

There are a number of different implementations of allocators, which roughly fall into the categories: sequential fit, segregated fits, and buddy systems. **Johnstone and Wilson**[8] give an excellent explanation of these implementations, which is repeated and paraphrased below in sections 2.2.1 to 2.2.3.

2.2.1 Sequential fit algorithms

These are the simplest of all allocators – all the free blocks are held on a single linked list. In fact, an even simpler one is described in [10], where allocations take place sequentially from an area of memory (actually an array), and deallocations can only take place in reverse order.

`free(3)` either places feed blocks on the front of the linked list (the "free list"), or places them in the free list such that all free blocks are held in address order (either increasing or decreasing). `malloc(3)` searches for an appropriate free block either from the front (LIFO) or from the end (FIFO)

of an unordered list, or from the front (address ordered – AO) of an address-ordered free list. These allocations can be performed as first-fit, next-fit, best-fit and worst-fit (see below). Extra optimisations can be applied to the basic algorithm[9], such as coalescing adjacent free blocks (to minimise fragmentation), and, if the freed block is on the top of the heap, then that chunk can be returned to the kernel, thus keeping heap (and hence total memory) usage as low as possible.

First-fit

The first fit mechanism searches through a free list from the start of the list until a block that is large enough to satisfy the allocation request can be found. When this happens, then, if there is enough excess space to create a new free block, one is created and added to the free list, and a pointer to the data area of the just-allocated block is returned. If there is no block large enough, then more memory is requested from the kernel, and the allocation retried.[5]

Next-fit

This mechanism is identical to operation to the first fit algorithm, with the exception that it uses a roving pointer for allocation [13]. The pointer records the point where the last search was satisfied, and the next search begins from there. If the pointer wraps around the free list, and the request was not able to be satisfied, then more memory is requested. This allocator is the one described in [11]. In theory, this allocation method should result in having fewer small unusable blocks at the start of the list[13]; however, practically, it has been found[8] to actually increase fragmentation.

Best-fit

A best-fit allocator searches through the entire free list for the smallest block that is able to satisfy a request. This helps ensure that fragments are as small possible. Knuth[14] thought that best-fit would create too many small fragments, and worst fit (see below) would provide better fragmentation performance; however practical results in [8] have shown that this policy tends to be best with regards to fragmentation. However, due to needing to search the free list, it can be slow if there are many items on the list, and hence a sequential best-fit policy is very rarely used.

[5]This is the usual behaviour. Some implementations may only work with a fixed size heap, and simply return an out of memory error, instead of requesting more memory.

Worst-fit

A worst-fit allocator searches the entire free list for the largest block possible, if an exact size block cannot be found (depending on implementation – some implementations will not check for an exact fit first). The reasoning is that if large blocks are split off, the remaining amount is more likely to be able to still satisfy a request.

2.2.2 Segregated free lists

A segregated free list allocater uses a set of free lists, where each list holds blocks of a particular size. When a request is made, the appropriate free list is used to satisfy the request. When a block of memory is freed, it as added to the free list for that particular block size. The use of multiple free lists makes the *implementation* faster than searching a single linked list. However, what is often *not* appreciated when discussing merits of segregated lists[8] is that *policy* is also affected in an important way – using the segregated lists means that good fits are usually found, so policy tends to be good-fit or even best-fit strategy, despite often being described as first-fit.

There are a few important variations on segregated free lists:

Segregated free lists

In this variant, each set of free lists holds a range of free blocks ranging from the particular class size up to the next larger size. In each particular list, a first-fit or next-fit policy tends to be used.

Exact-fit segregated free lists

This imposes an extra constraint on the standard segregated free list – each free list is constrained to holding blocks of only one size. This would be more likely to provide an exact-fit for a requested size compared to a standard segregated free list; however, memory overhead is increased as there need to be a larger number of free lists in memory, of which a number are likely to be empty.

Simple Segregated Storage

In this variant, larger free blocks are not split to satisfy requests for smaller sizes, and smaller free blocks are not coalesced to satisfy requests for larger sizes. If a request comes in for a particular size, and the size class is empty, more storage is requested from the operating system. Typically, one or two

virtual memory pages are requested at a time, and split into same-sized blocks which are then put on the free list. Since each page contains only one size of block, this method is called simple segregated storage.

2.2.3 Buddy systems

Buddy systems [12][15] are a variant of segregated free lists, supporting a limited, but efficient, kind of splitting and coalescing. In simple schemes, the entire heap is conceptually split into two large areas which are called *buddies*. These areas are repeatedly split into two smaller buddies until a sufficiently small chunk is achieved. This hierarchical division of memory is used to constrain where objects are allocated and how they may be coalesced into larger free areas. A free area may only be merged with its buddy, the corresponding block at the same level in the hierarchical division. The resulting free block is therefore always one of the free areas at the next higher level in the memory-division hierarchy. At any level, only buddy pairs may be merged. The purpose of this constraint on coalescing ensures that when a block is freed, its buddy can always be found by simple address computation, and the buddy will always be entirely free or unavailable (an unavailable chunk may have been split and have some of its sub-parts allocated, but not others).

Two significant variations on the buddy systems are outlined below.

Binary Buddy

This is the simplest and best-known of the buddy systems [12]. All buddy sizes are a power of two, and each size is divided into two equal parts. This makes address computations simple, as all buddies are aligned on a power-of-two boundary offset from the beginning of the heap area, and each bit in the offset of a block represents one level on the buddy system's hierarchical splitting of memory – if the bit is 0, it is the first of a pair of buddies; if the bit is 1, it is the second. These computations can therefore be implemented very efficiently with bitwise logical operations.

However, due to the requirements that any object size needs to be rounded up to the next power of two, internal fragmentation can be quite high (around 25%[8]).

Double buddy

A double buddy system uses two different buddy systems with staggered sizes. For example, one system may use power-of-two sizes (2, 4, 8, etc.),

while the other uses power-of-two spacing starting at a different size, such as 3, resulting in sizes 3, 6, 12, 24, etc. Request sizes are rounded up to the nearest size class in either series. This reduces the internal fragmentation by about half, but means that a block of a given size can only be coalesced with blocks in the same size series.

2.3 Symbian OS and smartphones

Symbian, established in 1998, is a software licencing company that develops and supplies their own operating system for data-enabled mobile phones (smartphones). As of December 2003, 18 phones from five manufacturers are using this OS[6].

The term "smartphone" is used to describe a mobile telephone device which is becoming more capable as a computing device; they tend to encompass the functionality of PDAs, personal organisers, address books and telephony. These are still, however, low-memory devices (typically varying between 8MB and 64MB of RAM), and hence using memory efficiently is essential to the performance of these devices (see chapter 1).

2.3.1 Symbian OS background

The Symbian OS, in its current form, has been around for a few years. A new operating system (named EKA2) is in current development, and is regarded as a real-time OS, thus making it considerable more suitable for modern and future smartphones that require some guaranteed response time from the kernel (for applications such as video streaming); this is currently not scheduled for release for approximately another year from time of writing. Nevertheless, for the purposes of this project it provides some useful features above the old OS (named EKA1), such as extensibility of the memory manager (see section 3.4), the testing of which is one of the aims of this project.

As previously mentioned, the Symbian OS runs on various phones, which fall into roughly three categories[2] (see below). The text below, until section 2.4 (not inclusive) is taken and (partially) paraphrased from [2], [3] and [4].

Mobile phones with a numeric keypad

These phones are designed for one-handed use and require a flexible user interface (UI) that is simple to navigate with a joystick, jogdial, or similar

[6] `http://www.symbian.com/about/about.html`

input method. The best example is the *Series 60* platform from Nokia, which forms the basis of phones such as the Nokia 7650, 3650 and N-Gage.

Mobile phones with touch screens

These tend to have larger screens than the first category, and dispense with the numeric keypad. The larger screen is ideal for working on the move and uses pen-based interaction with the user. The best current example of this form factor is *UIQ*, which is the platform for the Sony Ericsson P800. (The P800 actually provides a detachable numeric keypad which sits over the bottom part of the screen and allows the phone to be used in a 'traditional' way).

Mobile phones with a full keyboard

These mobile phones have the largest screens of all Symbian OS phones, have a full keyboard and could also include a touch screen (so the devices is more like a PDA than a proper phone). This best example is the *Series 80* platform from Nokia; this UI is the basis of the Nokia 9200 series, which has been used in the 9210 and 9210i.

2.3.2 Symbian OS naming conventions

Like any system, the Symbian OS has its own naming conventions to indicate what is important. A complete list and description may be found in [3], but the conventions that are relevant to this project are given below.

Fundamental types

The Symbian OS defines a few definitions for 8-, 16- and 32-bit integers and some other basic types that map onto underlying C++ types such as `unsigned int`, and which are guaranteed to be the same regardless of C++ implementation (which is important given that the OS runs on multiple architectures, and can use different compilers). Some of the definitions include:

- `TInt16` and `TUint16` – signed and unsigned 16-bit integers
- `TInt32` and `TUint32` – signed and unsigned 32-bit integers
- `TInt` and `TUint` – signed and unsigned integers: in practice, this means a 32-bit integer

- `TBool` – Boolean; actually equated to `int` due to the early compilers that were used
- `TAny` – equated to `void`, and usually used as `TAny*` (a 'pointer to anything')

Class names

Classes use an initial letter to indicate the basic properties of the class. Relevant ones for this project are:

- T classes, types (e.g. `TDesC`) – T classes don't have a destructor. They act like built-in types, which is why the `typedef`s for all built-in types begin with `T`. `T` classes can be allocated as automatics (if they're not too big), as members of other classes, or on the heap.
- C classes – Any class derived from `CBase`. `C` classes are *always* allocated on the default heap. The `new` operator initialises all member data to zero when an object is allocated. `CBase` also includes a virtual destructor, so that by calling `delete` on a `CBase*` pointer, any `C` object it points to is properly destroyed.
- R classes (e.g. `RHeap`) – Any class that owns resources other than on the default heap. Usually allocated as member variables or automatics; in a few cases, can be allocated on the default heap.
- Static classes (e.g. `User`) – A class consisting purely of `static` functions that can't be instantiated into an object.
- Structs – A C-style `struct`, without any member functions.

Data (variable and constant) names

As with class names, these also use an initial letter to identify the type:

- Enumerated constant (e.g. `EFalse`, `ETrue`) – constants in an enumeration (a logical set)
- Constant (e.g. `KMaxFileName`) – constants of the `#define` or `const TInt` type. `KMax`-type constants tend to be associated with length or size limits
- Member variable (e.g. `iTopMostCell`) – any nonstatic member variable. The `i` prefix refers to an 'instance' of a class

- Arguments (e.g. `aSize`, `aCell`) – any variable declared as an argument (the `a` stands for 'argument')
- Automatics (e.g. `x`, `i`) – any variable declared as an automatic

Function names

Here, the final letter marks the type of function:

- Non-leaving function (e.g. `Alloc()`, `Free()`) – Use an initial capital
- Leaving function (e.g. `CommitL()`) – Use final `L`

A leaving function may need to allocate memory, open a file, and so on – generally, to do some operation that might fail because there are insufficient resources of for other environment-related conditions (not programmer errors). When a leaving function is called, consideration must always be given to what happens when it succeeds and when it leaves. Both cases must be handled (Symbian OS's cleanup framework is designed to allow this to be done). This is the Symbian OS's most important naming convention.

Leaving functions are basically a lighter-weight version of C++ exceptions. They provide a way to unwind a call stack to a known good state, and with the help of the cleanup stack, cleaning up allocations on the way. The cleanup stack addresses the problem of cleaning up objects that have been allocated on the heap, but to which the *only* pointer is an automatic variable. If the function that has allocated the objects leaves, the objects need to be cleaned up. Since Symbian OS does not use C++ exceptions, the cleanup stack is its own mechanism to ensure that this cleanup happens. For further details on the cleanup stack, refer to [5].

2.3.3 Compilation for Symbian executables

Project specification file

Building an executable for the Symbian OS requires a number of steps. Initially, each set of source code needs to have a project specification file, which is then translated by the supplied tools into the makefiles or project files for one or more of the possible build environments. Project specification files have a `.mmp` extension (which stands for 'makmake project'). An example, taken from [4] is:

```
// hellotext.mmp
TARGET          HelloText.exe
```

```
TARGETTYPE      exe
SOURCEPATH      .
UID             0
SOURCE          hellotext.cpp
USERINCLUDE     .
SYSTEMINCLUDE   \epoc32\include
LIBRARY         euser.lib
```

The `TARGET` specifies the executable to be generated, and the `TARGETTYPE`, in this case, confirms it is an `.exe`. The `UID` is irrelevant for `.exe` files; however, it is necessary when building DLLs[7]. `SOURCEPATH`, `USERINCLUDE` and `SYSTEMINCLUDE` all specify directories to be searched for source files, user include files (specified on a `#include` directive with quotes), and system include files (specified with angle brackets on a `#include` directive). All Symbian OS projects should specify, at the least, `\epoc32\include` for their `SYSTEMINCLUDE` path. The `LIBRARY` directive specifies libraries to link to – these are the library files corresponding to the shared library DLLs whose functions are called at runtime.

Component defition file

The Symbian OS build tools require a component definition files to be present. This file always has the name `bld.inf` and contains a list of all the project definition files that make up the component. An example, again taken from [4] is:

```
// BLD.INF
PRJ_MMPFILES
hellotext.mmp
```

Compilation from the command line

Once the project specification and component definition files have been installed, it should be possible to build an application on the command line. Opening up a command prompt and changing to the directory containing the build files, the first command that needs to be run is:

```
bldmake bldfiles
```

[7] A set of unique UIDs is normally allocated to the programmer by Symbian. This is most important if the finished programs are released; uniqueness does not need to be guaranteed for internal-only development

This command should return without displaying anything – it creates a file called `abld.bat` which is used to drive the remainder of the build process.

`abld.bat` provides a number of important targets that are used in the whole build process. These are run with `abld <target>`, where the targets are outlined below:

- `build` – this builds the source code into a binary, and places the binary file into its appropriate place (when using the emulator)
- `clean` – this cleans out object code and temporary files
- `reallyclean` – this cleans out object code, temporary files, and also the binaries that were created (so essentially restores the system to a state before any compilation has taken place)
- `freeze` – this assigns ordinal values to any exported functions (see section 3.2) and creates a `.def` file containing the exported functions' ordinal values.

All of these targets can optionally be followed by a target architecture and build type – e.g. `abld build winscw udeb` will build the code for the emulator (using the CodeWarrior compiler), using only the debug build `udeb` (the `u` means the build uses a Unicode character set). If the build type is not specified (e.g. just `abld build winscw`), then the code will be built for both the debug build and a release build (specified with `urel`).

More on the command-line build process is found in [4].

Compilation under CodeWarrior

To compile under CodeWarrior IDE (one of the standard build environments at Symbian), the project first needs to be imported into the IDE. This is achieved by selecting `File | Import Project from .mmp File` from within the IDE, selecting the appropriate SDK and then selecting the appropriate `.mmp` file to import. Compilation of the source code is achieved by selecting `Project | Make` or pressing F7 – the default target is `winscw udeb`. The Symbian OS emulator can be launched by selecting `Project | Run` or pressing Ctrl-F5, or the code can be run through the debugger by selecting `Project | Debug` or pressing F5.

2.4 Johnstone and Wilson paper

This paper[8] presents a simulation study on various memory allocation policies, using memory allocation traces from real programs. A goal of the re-

search was to measure the true fragmentation costs of particular memory allocation policies independently of their implementations. This paper is widely regarded as a canonical study of memory allocator implementations when running with a range of real-world applications, and has a large number of citations from subsequent memory allocator studies.

2.4.1 Experimental design

The test programs were linked into a malloc trace-gathering library, and a trace for each program was generated. The trace contained information about the type of operation performed (i.e. malloc, free or realloc), as well as the locations of the cells where these operations were being performed, and the number of bytes requested in each malloc or realloc. For a realloc operation, the relocation address was also recorded, so a reallocated cell could be kept track of where it had moved from and to.

Having obtained traces files for the memory request patterns of the test programs, these traces were then used as an input to test each allocation policy. The processing involved reading an operation from the trace file, and calling the appropriate procedure in the particular allocator implementation that was being investigated at that time. Additionally, the amount of memory allocated and freed by each allocator was also recorded after processing each operation in the trace file, as well as the number of bytes requested from the operating system by the allocator to be able to service the request. This data would be used in measuring fragmentation.

2.4.2 Measuring fragmentation

All the allocation policies investigated by [8] used the technique of splitting to always return a cell of exact size as requested, and so produced zero internal fragmentation (see chapter 1 for discussion on internal and external fragmentation), hence the only form of fragmentation that could occur in the experiments was external fragmentation.

To measure this fragmentation, two values were kept track of when processing the trace file – the amount of live memory requested by the program and the amount of memory used by the allocator to satisfy the program's memory requests. From these, a measure of the amount of fragmentation occurring could be calculated. Figure 2.2 shows how these values vary when using a simple segregated storage allocation policy for a particular program.

The paper describes how fragmentation could be measured in any one of four different ways; the two methods that they actually chose to use were as follows:

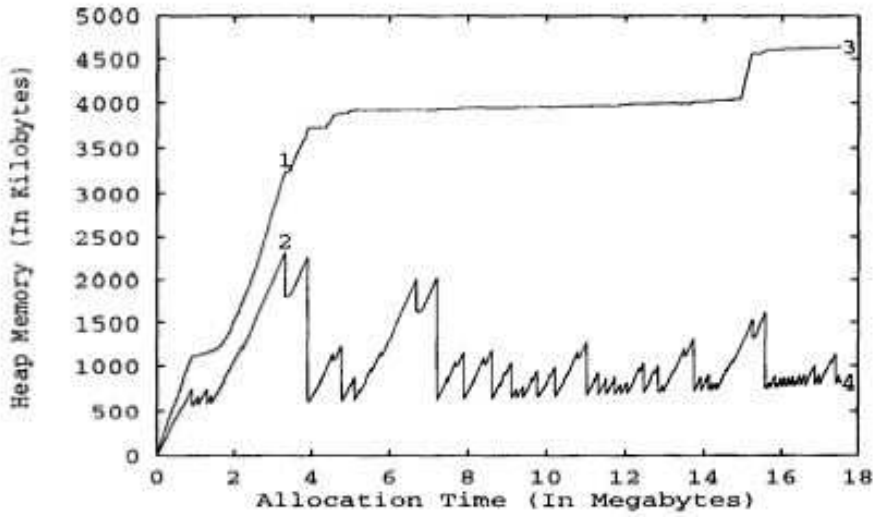

Figure 2.2: Measurements of fragmentation for a gcc test program trace using a simple segregated storage allocation policy; figure taken from [8]

Fragmentation measurement: method 1

The maximum amount of memory used by the allocator relative to the amount of memory requested by the program *at the point of maximal memory usage*. In figure 2.2 this corresponds to the amount of memory at point 3 relative to the amount of memory at point 4.

The report states that the problem with this measure of fragmentation is that it will tend to report high fragmentation for programs that use only slightly more memory than they request if the extra memory is used at a point where only a minimal amount of memory is still live.

Fragmentation measurement: method 2

The maximum amount of memory used by the allocator relative to the maximum amount of live memory. In figure 2.2 this corresponds to the amount of memory at point 3 relative to the amount of memory at point 2.

The report states that the problem with this measure of fragmentation is that it can give a number that is too low if the point of maximal memory usage is a point with a small amount of live memory, and is also the point where the amount of memory used becomes problematic.

2.4.3 The results

Eight test programs were used and trace files of their memory request patterns generated. These trace files were then processed in turn with various different allocation policies and the fragmentation results averaged across all the programs. Figure 2.3 shows the final results from the study.

Allocator	Average	Average
best fit AO 8K	0.98%	0.83%
best fit AO	3.73%	2.28%
best fit LIFO	3.76%	2.30%
best fit FIFO	3.93%	2.23%
first fit AO 8K	0.91%	0.78%
first fit AO	6.63%	2.30%
first fit LIFO	50.7%	36.3%
first fit FIFO	4.97%	3.14%
next fit AO	13.7%	8.04%
next fit LIFO	52.9%	38.4%
next fit FIFO	20.9%	18.4%
Lea 2.6.1	3.80%	2.28%
binary buddy	59.8%	53.4%
double buddy	36.2%	34.3%
simp seg 2^N	1818%	73.6%
simp seg $3 * 2^N$	1468%	61.5%
Average		
Std Dev		

Figure 2.3: Percentage fragmentation for various allocators averaged across all programs. The second column is the fragmentation measure using method 1 (section 2.4.2). The third column is the fragmentation measure using method 2 (section 2.4.2). Figure obtained from [8]

It can be seen that the two best allocation policies, first-fit address-ordered free list, and best-fit address ordered free lists, both using 8K memory requests from the operating system (instead of the standard 4K), have, on average, less than 1% actual fragmentation, using both measures of fragmentation[8]. Conversely, the allocation policies that did not coalesce all possible blocks produced higher fragmentation measures. With an 8K memory request, any extension to the heap requested by the allocator to the operating system (using the `sbrk(2)` system call) is rounded up to the next multiple of 8 kilobytes. A 4K request (which is usual in most allocator implementations, as this tends to be the size of one memory page) only rounds the request to the next multiple of 4 kilobytes.

[8]In fact, within the bounds of experimental error, the best performing allocators have about the same level of fragmentation

The paper concludes with a statement that, for a large class of programs, the fragmentation "problem" is really a problem of poor allocator implementations, and for these programs, well-known policies suffer almost no true fragmentation.

2.5 Bohra and Gabber paper

This paper[1] presents a study of the dynamic memory activity pattern of Hummingbird[9] and GNU emacs[10]. It compares the behaviour of nine different `malloc` implementations when used with the dynamic memory activity traces. The main goal was to measure the fragmentation of the different implementations; however a secondary goal was to produce statistics of allocated object sizes, such as the relation between number of objects and size of objects allocated.

2.5.1 Experimental design

Both Hummingbird and GNU emacs were instrumented to generate a trace of their dynamic memory activities. The trace itself was a text file, in which each line corresponded to either a memory allocation or deallocation operation, with each line additionally having a tag, so it would be possible to match allocation and free operations (so the correct pointer would be fed to the `free` routine when running the data against real `malloc`s.

The Hummingbird trace corresponded to the processing of an HTTP proxy log of four days. The trace contained about 58 million events, which corresponded to the dynamic memory activity of Hummingbird when it was processing 4.8 million HTTP requests. The requests contained 14.3GB of unique data, and 27.6GB total data.

The emacs trace was obtained when running an emacs macro that listed the contents of `/usr/src/` using the `dired` directory editing mode, visited all the `.c` and `.h` files found, and changed the string `int` to `uint32` in all files. The files were all edited sequentially by emacs. The particular directory used contained a large number of Linux kernel source trees, with 73,212 `.c` and `.h` files, totalling 2.7GB. The dynamic memory activity trace contained about 20 million memory allocations and deallocations.

[9]Hummingbird is a light-weight file system for caching web proxies, which was designed specifically to improve the access time of caching web proxies to cached objects stored on the disk, specially when compared to the standard Unix file system which is not very well suited to this application

[10]This was picked due to its wide use, and emacs sessions tend to be run for an extended period of time as a main working environment

Having obtained the traces, a simple driver program was written which read the trace and called the corresponding `malloc` or `free` based on the current trace file entry.

2.5.2 Measurements

The heap sizes of nine `malloc`s were measured when using the same Hummingbird and emacs dynamic memory access traces; most of the `malloc`s that were used were open source packages, two of them were implementations used in Solaris. Fragmentation percentage was computed by using $100 \times \left(\frac{heapsize}{live\ memory} - 1\right)$ (where *heapsize* and *live memory* are the maximum heap size and maximum live memory respectively). In addition to fragmentation, the distribution of object sizes was also recorded.

2.5.3 Results

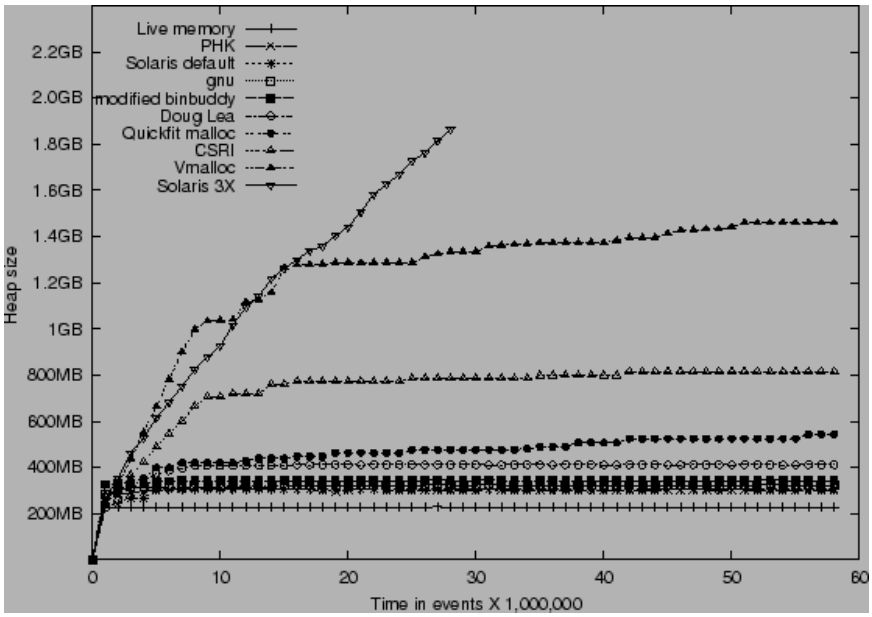

Figure 2.4: Comparison of the heap size of nine `malloc`s when used with the Hummingbird dynamic memory activity trace. The *Live memory* line shows the actual amount of live memory. Figure obtained from [1]

Figure 2.4 shows the heap size when running the Hummingbird trace. The best allocator (least fragmentation) turned out to be one written by

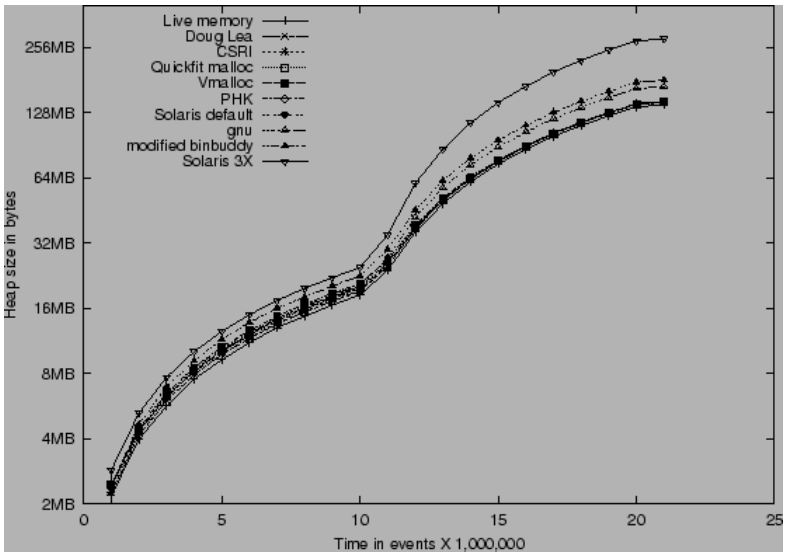

Figure 2.5: Comparison of the heap size of nine `malloc`s when used with the emacs dynamic memory activity trace. The *Live memory* line shows the actual amount of live memory. Figure obtained from [1]

Poul-Henning Kamp and used in the BSD operating systems[11] with 30.5% fragmentation, whereas the worst allocator was a "space efficient" allocator distributed with SunOS 5.8, which caused a heap overflow.

When running the emacs trace (figure 2.5), the best allocator turned out to be one written by Doug Lea[12] with 2.69% fragmentation. The worst allocator, again, turned out to be the "space efficient" allocator with 101.48% fragmentation.

Comparing the distribution of object sizes shows two completely different patterns between Hummingbird and emacs – the distribution of emacs object sizes is fairly random (figure 2.6), whereas the distribution of Hummingbird object sizes (figure 2.7) is, for objects larger than 128 bytes, almost a straight line on the log-log scale.

The paper concludes by stating that the results obtained should hopefully renew interest in dynamic memory allocation, and lead to better understanding why particular dynamic memory allocation schemes work better for the kind of dynamic memory activity described in the paper. Also, future `malloc` implementers should consider dynamic memory activity patterns similar to

[11] Source code is available from
`ftp://ftp.freebsd.org/pub/FreeBSD/src/lib/libc/stdlib/malloc.c`

[12] Source code is available from
`http://gee.cs.oswego.edu/pub/misc/malloc.c`

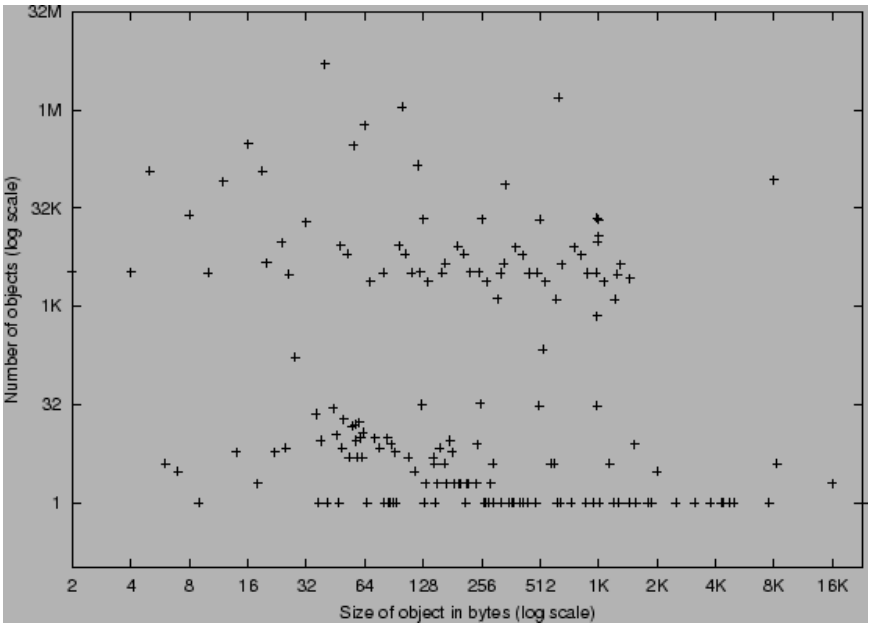

Figure 2.6: Distribution of emacs object sizes

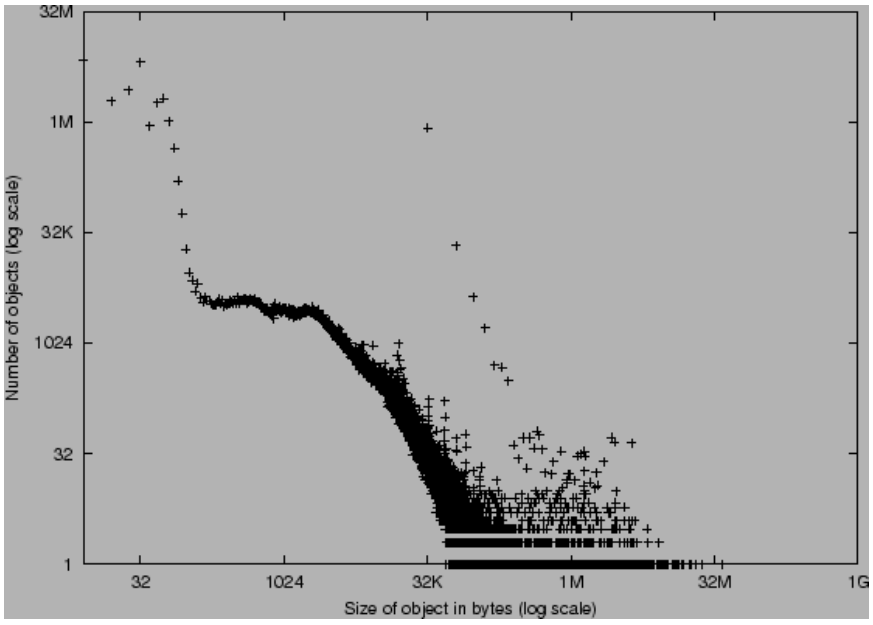

Figure 2.7: Distribution of Hummingbird object sizes

Hummingbird's when updating their code and, at the minimum, application developers should become aware of excessive memory fragmentation problems, and, if they occur, should attempt to alleviate it by picking a different `malloc` package.

Chapter 3

Review of Symbian OS framework

Section 3.1 and section 3.2 are taken and paraphrased from [6], and given as background information to Symbian OS framework. Sections 3.3 onwards are part of the investigation into the operation and extensibility of the Symbian allocator, and contain some details previously undocumented.

3.1 Extensibility with DLLs

A Symbian OS DLL[1] is a library of program code with potentially many entry points, which separates it from a Symbian OS `.exe` file, which has a single main entry point, `E32Main()`.[6] When the Symbian OS launches a new `.exe`, it first creates a new process, and then calls the entry point in the context of the main thread of that process. A DLL, on the other hand, is loaded into the context of an existing thread.

There are two important types of DLLs – a shared library DLL, and a polymorphic DLL. A shared library provides a fixed API that can be used by one or more programs. Executables are marked with the shared libraries that they require, and the system loads the shared libraries along with the executable at run-time automatically. Shared libraries are loaded recursively, so if a particular loaded library requires others to be loaded, those will also be loaded, until everything required by the executable is ready.

A polymorphic DLL is more useful for extending the operating system itself – it implements an abstract API such as a printer driver, sockets protocol, or an application.[2] Polymorphic DLLs usually have a single entry

[1] Dynamic Link Library

[2] In the Symbian OS, such DLLs typically use an extension other than .dll

point, which allocates and constructs a derived class of some base class associated with the DLL. Polymorphic DLLs are usually loaded explicitly by the program that requires them.

3.2 DLL and executable operation in the Symbian OS

If an executable is run from ROM (i.e. it is built into the system and appears as drive Z: in the OS), then it is executed in-place – it does not get copied into RAM. Executables that are on the system flash RAM drive (C:) or removable media, however, are first copied into RAM before being executed.

A .exe file in Symbian is not shared – if it is loaded into RAM, it has its own areas for code, read-only data and read/write data.[3] If a second version of the same .exe is launched, a new area of memory will be allocated. As a small optimisation, ROM-based .exes allocate a RAM area only for read/write data – the rest of the binary data is read directly from ROM.

DLLs, however, are shared – when a DLL is first loaded, it is relocated to a particular address. When another thread requires the same DLL, it merely attaches the copy already there, so it will still appear at the same address to all threads that use it. The OS maintains reference counts so that the DLL is only unloaded when no more threads are attached to it. ROM-based DLLs are never loaded into RAM – they are simply used in-place, like ROM-based .exe files.

The OS also optimises the formats used for DLLs in order to make them as compact as possible: most systems supporting DLLs offer two options for identifying the entry points in them – one is to identify the entry points by name, the other is by ordinal number (each entry point is identified by a particular number). Names are potentially long (and hence wasteful of memory), so the Symbian OS only uses link-by-ordinal exclusively. However, this means that if the exported functions are moved in the file[4], any executables linking to that DLL will need to be recompiled to fit with the new ordinal scheme, and the exported functions will need to be frozen (this gives each function its particular ordinal number, see also section 4.2.2).

Loading a DLL into RAM involves locating the executable at an address that is not determined until load time, which means that relocation information must be included in the executable format. However, for a file in ROM,

[3] All executables contain program code, read-only static data and read/write static data under Symbian

[4] If any DLLs are released to third-parties to use, then it is fairly essential that future releases use the same ordinals as previous releases

which is executed in place, the address is always known, and hence the Symbian OS ROM-building tools automatically place the DLL in a known place, and strip out the relocation information to make them still smaller.

3.3 Operation of the memory allocator

The Symbian memory allocator implementation is contained in a class called `RHeap`. However, the interface to the allocator is provided by a static class called `User`[5], specifically the methods `User::Alloc()`, which behaves like C's `malloc()`; `User::Free()`, which behaves like C's `free()`; and `User::Realloc()`, which behaves like `realloc()`. Leaving variants (see section 2.3.2), such as `User::AllocL()`, are also provided.

Heaps for processes are created in `Chunks`. A Chunk is an area of memory (the same size as the maximum heap size) which may be local or global. A local chunk is private to the process creating it and is not intended for access by other user processes. A global chunk is one which is visible to all processes. To create a heap, a function named `UserHeap::ChunkHeap` needs to be called – this creates a local or global chunk (if a named Chunk is given as a parameter to the function, a global chunk is created; if that parameter is `NULL`, then a local chunk is created), and creates a heap inside that chunk (the minimum and maximum sizes of the heap are also given as parameters to the `ChunkHeap` function; the size of the chunk is set to the maximum size of the heap).

The `RHeap` code implements a first-fit sequential memory allocator with coalescing, maintaining the free list in address order. An overview of the allocation, freeing and reallocation procedures follow:

3.3.1 Allocation

When a block of a particular size is requested, it is initially rounded to the maximum value of either the minimum block size, or the requested size plus the size of the header of an allocated block, at which point an attempt is made to allocate it. If this allocation fails, then an attempt is made to extend the heap, and allocate again. If this then fails, a `NULL` pointer is returned. If the allocation succeeds then a pointer to the start of the data segment of the block is returned; this ensures that the block header which contains important information such as the size of the block will not be overwritten.

[5]This class provides more than 70 functions in various categories; more information is found in the Symbian API reference, provided with development kits

When actually allocating a block on the heap, the allocator scans down the free list until it finds a block that is large enough to hold the requested size. If the difference between this block size and the requested size is greater than or equal to the minimum allocatable block size, the excess is split off, and the new block created from this splitting is added in the free list in the same place where the just-allocated block was. If the difference is smaller than the minimum block size, the entire block is used for the allocation. If the end of the free list is reached (no allocation was possible), then a `NULL` pointer is returned, otherwise a pointer to do newly allocated block is returned.

3.3.2 Freeing

If a `NULL` pointer is passed into this function, then the function does nothing; otherwise, it places a previously allocated block back on the heap. No check is made that the block has not been prevoiusly freed (undefined results may happen if this occurs). When placing a block back on the heap, the linked list is first scanned down to find if there is a free block in the heap directly following the to-be-freed block. If there is, then the new block is coalesced with the next block. The preceeding free block is then checked to see if it is adjacent in the heap to the new block; if it is, then the new block is additionally coalesced with it. If the coalescing cannot be done, then the new block is inserted into the free list in between the block with a lower address and the one with a higher address, thus the free list is always maintained in address order.

If the freed block (after any coalescing) is at the end of the heap, and its length is greater or equal to twice the minimum amount of memory requested by the allocator when growing the heap (by default 4 kilobytes), then an attempt is made to reduce the heap size (returning memory to the kernel to be able to be used by other applications).

3.3.3 Reallocation

The requested size of the block to be reallocated is firstly rounded to the maximum value of either the minimum block size, or the requested size plus the size of the header of an allocated block. If this size is greater than the current length of the block, then initially an attempt is made to grow the block in place (i.e. not moving it). If this is successful, then a pointer to the current block is returned. If not, then a new block is allocated (if this allocation fails, then a `NULL` pointer is returned), the data copied from the old block to the new block, up to the length of the old block: the extra memory can hold any arbitrary value (i.e. should not be assumed to hold zeros (null

values)), and the old block is then freed up. A pointer to the new data block is returned.

If the requested size (after rounding) is smaller than the current block size then, if this size difference is greater than or equal to the minimum block size, a new block is created and linked into the free list. The pointer returned from the reallocation procedure then still points to the same memory block (i.e. a reduced-size block is never moved).

3.4 Extensibility of the memory allocator

The Symbian `RHeap` class extends a class called `RAllocator`, which in turn extends a base class called `MAllocator`. This class contains declarations of the allocator methods (`Alloc`, `Free`, etc.) which are marked with the C++ keyword `virtual`. This keyword indicates that any methods with those names in an inherited class will override the methods in the class that is inherited from.

The `RAllocator` class itself does not define any allocator methods: this is left to the `RHeap` class, whose definitions of the allocator methods are also virtual. This is used when implementing the allocator tracing (see chapter 4) as a derived class from `RHeap`, as the tracing functions override the `RHeap` methods in a heap created from this derived class.

Additionally, the OS provides methods to switch to different heaps or allocators, accessed from the `User` class. One method is `User::SwitchHeap()`, which takes as its parameter an `RHeap` object (i.e. heap), or one inherited from `RHeap`, and returns a pointer to the old heap. The other method provided is `User::SwitchAllocator()`, which takes as its parameter an `RAllocator` derived object (i.e. heap), and returns a pointer to the old heap.

Both of these methods allow use of different allocator implementations to the standard Symbian implementation; the extent of how well this extensibility works is investigated in chapter 6.

Chapter 4

Design and Implementation

This chapter will give the design and implementation of extensions to the Symbian allocator to enable trace file generation (the implementation will discuss important aspects of the code). It will also give some details as to how the trace files were processed. Issues encountered will also be discussed.

4.1 Design

4.1.1 Objectives

Characterising smartphone applications

Characteristics of the smartphone applications themselves are obtained by measuring the number of objects of a particular size across the whole run of a program. This parallels one of the sets of application characteristics given in [1].

Assessing the existing Symbian allocator

Assessment of the existing Symbian allocator is done by giving a measure of external memory fragmentation, paralleling the study of [8]. Given that there are multiple ways of measuring fragmentation, the methods chosen will also parallel [8] (see section 2.4.2).

4.1.2 Emulator

Most Symbian development is initially done using an emulator[7]. The emulator includes a number of applications and mimics a real Symbian OS phone very closely. The particular emulator interface that was used in the running

of applications is the interface used in the Series 80 platform for Nokia (see also section 2.3.1). Using the emulator for developing the allocator extensions allows for faster development compared to running on real hardware – the use of a built-in debugger in the development environment, for example.

4.1.3 Experimental Design

The benchmark applications

The applications selected are a range of smartphone applications that would be used in day-to-day smartphone use, such as an Agenda, Alarm, Address book and Communications program. The latter is used to send emails and text messages; unfortunately the functionality to actually send and receive data was not present on the system (requiring a modem), so any messages had to be saved on the local machine without being sent.

Trace information to be gathered

For the purposes of analysing the performance of the Symbian memory allocator, and to measure overheads, the following traces need to be obtained:

- Operation (whether the operation is an Alloc, Free or ReAlloc operation)
- Number of bytes requested
- Number of bytes allocated
- Address of the allocated memory (this can then be used to match a Free operation to its conjugate Alloc operation)
- Start time of the operation
- End time of the operation
- Heap size at the end of the operation

Do to the operation of the Symbian allocator when performing reallocations (see section 3.3.3), a reallocation operation is only recorded if the block of memory that is being reallocation does not move in the heap. If the block is moved, this is recorded as an allocation followed by a free operation.

The timing information is there to be able to compare the speed of performing the operations (specifically the time taken for each operation) under the Symbian allocator, with the same operations under a different allocator

(see section 6.1). The output of bytes allocated and bytes requested give a measure of how much space is wasted due to over-allocation of cells, and outputting the heap size will give a measure of memory fragmentation, when compared to the live data used; this then can be used to assess the allocator. The total live data can be calculated from the traces when processing the data, and hence does not need to be recorded.

To characterise the smartphone applications, a record of the number of allocations of an object of each particular size can be used (thus obtaining a frequency distribution of object sizes). The number of objects can be calculated when processing the data, hence no record of this needs to be kept as part of the tracer, only the size of the object.

Method

Due to the extensible nature of the Symbian OS, the tracing was implemented as a class called `MyHeap` that extended the existing Symbian allocator. The new class was compiled as a polymorphic DLL file (see section 3.1), so the Symbian OS would pick it up and automatically use it instead of using the built-in allocator (`RHeap`).

Each particular procedure (Alloc, ReAlloc and Free) needs to be overridden with an equivalent procedure that includes tracing.Constructors need to be specified, so initialisation is done correctly (calling through to the parent class). Additionally, all the methods that are used to create a ChunkHeap (see section 3.3) need to be re-implemented to use `MyHeap` instead of the Symbian `RHeap`.

To avoid having issues with the recording of the traces interfering with the allocation/deallocation of memory (by causing extra allocations), the records were stored as a large array of `structs`, where each `struct` contained enough information to be able to obtain an allocation profile (see section 4.1.3). This way, a non-heap area of memory is set aside for the traces.

Once the traces have been collected, it is necessary to have them written out to a file. For this, an extra function was used to go through the parts of the array that had been used to store traces, and write the `structs` out to a file (see section 4.2.3). Given that the Symbian OS has its own file system, it would be easiest to write the traces to a files in this filing system, which is also then accessible by the host system if running the emulator (see section 4.1.2). Additionally, unique names should be given to the trace files so they can be easily identified, and so that multiple programs (and hence traces) can be obtained in each session.

Due to the possibility of switching memory allocators in the Symbian OS (see section 3.4), each application should, in its startup code, switch to using

`MyHeap`, and, on shutdown, write out the traces and switch back to using the Symbian allocator. The startup routine that is executed when an application starts is `NewApplication()`[6], so the allocator switching should be implemented in this routine. This, of course, requires the modified application to be recompiled. Producing the traces would then be a matter of running the program and exiting it to obtain an allocator trace.

Processing of the trace files is to be done with an external program which reads in the trace files, and can then output statistics such as fragmentation measures, and values suitable for generating graphs of object size frequencies or memory sizes, when fed into a suitable graphing program.

Repeatability

It is necessary to be able to repeat the steps taken to obtain the traces so a fair comparison can be made between the results obtained here, and a future study. Due to the nature of the system being interactive in its use, it is not possible to generate a file or a script to be used as the input to a particular program; hence it was decided to record the steps taken when running the system, so this could be replayed and re-run.

Analysis of data

Once the data has been collected, it then needs to be analysed to be able to produce some meaningful information. For the purposes of this investigation, a measure of fragmentation (both internal and external) is required, as is a frequency distribution of object sizes.

4.2 Implementation

4.2.1 Project environment

Project specification file

To be able to compile the `MyHeap` code as a DLL file, the MMP (project specification, see section 2.3.3) file needed to be set up for building to a DLL target. This was done by setting the `TARGETTYPE` parameter to `DLL`. The `SOURCE` parameter only needed to contain one source code file, `myheap.cpp`. The full MMP file is given with other code at the end of the report.

Additionally, the component description file `bld.inf` needed to contain the following lines:

```
PRJ_MMPFILES
```

`myheap.mmp`

CodeWarrior

To build the extensions to the allocator, Metrowerks' CodeWarrior for Symbian OS was used, as this is one of the standard environments for Symbian development, and interacts with the emulator to provide full debugging capabilities, as well as placing the binary files in the correct place for the emulator to pick them up and use them.

vnc2swf

To have a record of how the applications were run, to ensure repeatable operations, a program called *vnc2swf*[1] was used. This captures a VNC[2] session to a Macromedia Flash[3] file. Unfortunately this doesn't allow for automatic replay of the actions taken – this still has to be done manually, repeating the actions as they are viewed from the Flash file.

The file generated from recording the traces is available on the CD included with the project as the file `\symbian.swf`. The file `\symbian.html` allows the file to be viewed from within a web browser.

4.2.2 The polymorphic DLL

Generating

The DLL itself is generated and put in place when compiling the code – the MMP file having specified that the target for compilation is a DLL file (see section 4.2.1).

Freezing, cleaning and rebuilding

Due to the way the Symbian OS handles DLLs, identifying entry points by ordinal, the exported functions need to be frozen (see section 3.2) before compilation can succeed. This is achieved by running the command `abld freeze` on the command line in the directory containing the source files (see section 2.3.3 and the system manual in appendix A). However, due to an apparent issue with the build system not picking up re-frozen exports, the entire source tree needs to be cleaned, re-frozen and rebuilt (this is covered in the system manual).

[1] `http://mail.unixuser.org/~euske/vnc2swf/`
[2] Virtual Network Computer, `http://www.realvnc.com`
[3] `http://www.macromedia.com/software/flashplayer/`

4.2.3 Code modifications to generate traces

This section will give an overview of the code used to implement the tracing. The entire code itself (which also includes code described in section 6.2) is given at the end of the report.

Overview

Given that the Symbian OS is written in C++, and the tracing is implemented as an extension to their allocator, the `MyHeap` code is also written in C++ (and is also hence split into a header file, and a file containing the main body of the source code).

To declare whether or not tracing should be used in a particular build of `MyHeap`, a C pre-processor defintion is used in the `MyHeap` header file – if tracing is required, then the line

```
#define tracing
```

is used; if tracing is not required, then this line is commented out.

A few useful macro definitions are also given, for example:

```
#define GETLENGTH(s)        ((s)->len)
#define SETLENGTH(s,l)      ((s)->len=l)
#define HEAPSIZE            ((TInt32)(iTop - iBase))
```

Although this is not strictly necessary here, using the macros rather than their definitions helps greatly with the implementation of a different memory allocator (see section 6.2). `HEAPSIZE` gives the size of the heap – `iTop` is the pointer to (i.e. address of) the top of the heap, and `iBase` is the pointer to the base of the heap.

The tracing information

To hold the traces, an array of `struct`s need to be set up. The code used is:

```
#define MAXTRACES 1000000

struct traceinfo
{
        TChar op; // Operation - alloc, free, etc.
        TInt32 request; // Number of bytes requested
        TInt32 alloced; // Number of bytes allocated
        TAny* address; // Address of operation
        TInt32 starttick; // System time at start (us)
```

```
        TInt32 endtick; // System time at end (us)
        TInt32 heapsize; // Heap size at end of operation
};

struct traceinfo trace[MAXTRACES];
TUint traceposition;
TInt32 minCellSize;
```

`MAXTRACES` represents the maximum number of traces that can be obtained – due to using an array, it needs to be given a particular size so that memory for it is pre-allocated and hence does not interfere with the trace gathering. `minCellSize` gives the value of the minimum allocatable cell size.

The variable `traceposition` is used to keep a record of the current position in the `trace` array – it is incremented each time an allocation, free or reallocation operation occurs.

MyHeap

Following the `struct` definition and the array in the header file is the class definition for `MyHeap` – this defines the public interfaces of the class (i.e. which need to be frozen), and the protected methods.

Given that `MyHeap` is an extension of the Symbian `RHeap` class, there is not a great deal in the main source code – simply the constructors for the class, and the routines outlined in the following sections.

The constructors call up to the `RHeap` constructors with the parameters that have been passed in, and then set the variable `minCellSize` to the smallest allocatable cell size. The `operator new` routine, which is called when a heap is created, performs a sanity check that the requested heap size is not smaller than the size of a heap class and sets up two of the instance variables of the heap class:

```
UEXPORT_C TAny* MyHeap::operator new(TUint aSize,
                                     TAny* aBase) __NO_THROW
/**
@internalComponent
*/
    {
    __ASSERT_ALWAYS(aSize>=sizeof(MyHeap),
                    HEAP_PANIC(ETHeapNewBadSize));
    MyHeap* h = (MyHeap*)aBase;
    h->iAlign = 0x80000000; // garbage value
    h->iBase = ((TUint8*)aBase) + aSize;
```

```
    return aBase;
    }
```

(This code was taken from the Symbian `RHeap` implementation, but references to `RHeap` were changed to `MyHeap`)

Alloc and Free

These routines write some trace information, and then call up to the Symbian implementations to perform the actual allocation or deallocation, and then write the rest of the trace information and increment the `traceposition` counter (assuming it hasn't reached `MAXTRACES`). The code for the Alloc routine is:

```
UEXPORT_C TAny* MyHeap::Alloc(TInt aSize)
    {
#ifdef tracing
    trace[traceposition].op='A';
    trace[traceposition].request=aSize;
    trace[traceposition].starttick=getTick();
#endif
    TAny* r = RHeap::Alloc(aSize); // Call Symbian allocator
#ifdef tracing
    trace[traceposition].endtick=getTick();
    trace[traceposition].alloced=GETCELLLENGTH((SCell*)r);
    trace[traceposition].heapsize=HEAPSIZE;
    trace[traceposition].address=r;
    if (traceposition<MAXTRACES)traceposition++;
#endif
    return r;
    }
```

The `UEXPORT_C` marks the procedure as one of the DLL entry points. As can be seen, the operation, the requested size and the time at the start is recorded, then the Symbian allocator is called. On return, the rest of the trace is recorded, before the procedure exits. It is worth noting that the address that is recorded is *not* the start of the cell in the heap, but the start of the usable data area, as returned to the calling procedure.

The `Free` routine is very similar, except for recording 'F' as the operation instead of 'A', and using -1 for the request size and amount allocated.

ReAlloc

The `ReAlloc` procedure is slightly more interesting – it may move the cell, or it may keep the cell in the same location, only changing its size. If the cell gets moved, then (as part of the Symbian implementation), a new cell gets allocated, and the old one gets freed up; the tracing for this is then taken care of by the Alloc/Free tracers. If the cell is not moved, however, then a record must be made of the `ReAlloc` operation. Note that this means the trace files lose some information about the pattern of reallocation operations, since these are often translated to an Alloc and a Free operation. Hence the code is slightly different to that in the section above:

```
UEXPORT_C TAny* MyHeap::ReAlloc(TAny* aCell, TInt aSize,
                                TInt aMode)
    {
#ifdef tracing
    trace[traceposition].starttick=getTick();
#endif
    TAny* r=RHeap::ReAlloc(aCell, aSize, aMode);
#ifdef tracing
    if (r==aCell)
    {
        // Cell hasn't been moved
        trace[traceposition].endtick=getTick();
        trace[traceposition].op='R';
        trace[traceposition].request=aSize;
        trace[traceposition].alloced=GETCELLLENGTH((SCell*)r);
        trace[traceposition].address=r;
        trace[traceposition].heapsize=HEAPSIZE;
        if (traceposition<MAXTRACES) traceposition++;
    }
#endif
    return r;
    }
```

DumpTrace

This procedure writes out the stored traces to a file, which can then be processed externally to generate useful results. The code for this is:

```
EXPORT_C void MyHeap::DumpTrace()
{
```

```
#ifdef tracing
    TUint i=0;
    RThread thread;
    TName threadName = thread.Name();
    TName name;
    name.Num(thread.Id());
    name.Insert(0,_L("_"));
    name.Insert(0,threadName);
    name.Insert(0,_L("C:\\traces\\"));
    name.Append(_L(".trace"));

    RFs file;
    RFileWriteStream tracefile;
    file.Connect();
    TInt err = tracefile.Create(file,name,EFileWrite);

    tracefile.WriteInt16L((TInt16)sizeof(SCell));
    tracefile.WriteInt32L(minCellSize);

    for (i=0; i<traceposition; i++)
    {
        tracefile.WriteUint32L((TUint32)trace[i].op);
        tracefile.WriteInt32L(trace[i].request);
        tracefile.WriteInt32L(trace[i].alloced);
        tracefile.WriteUint32L((TUint32)trace[i].address);
        tracefile.WriteInt32L(trace[i].starttick);
        tracefile.WriteInt32L(trace[i].endtick);
        tracefile.WriteInt32L(trace[i].heapsize);
    }

    tracefile.CommitL();
    tracefile.Close();
    file.Close();
#endif
}
```

The `EXPORT_C` marks the procedure as one of the DLL entry points (it is equivalent to `UEXPORT_C`). The `_L` macro creates a text literal from a given string (basically converting each character to its Unicode equivalent, as the Symbian OS uses Unicode internally).

Initially, the name of the trace file is constructed by obtaining the thread name and (numeric) process ID, prepending the directory to store the traces (C:\traces), and appending the extension `.trace`. Thus a complete trace filename would be `C:\traces\`appname_pid.`trace`. This naming was chosen to be able to generate filenames that were unique each time the emulator was run; additionally, this has the benefit of being able to see which application program has produced a particular trace file.

`RFs file` creates a new file server, which the Symbian OS uses to communicate with the file system; the connection is made using the `Connect()` method. `tracefile` is a streamed file writer (`RFileWriteStream`). When the file is created for output (using the `Create()` method), its mode is set to `EFileWrite`, which open the file for writing only, and causes an error if the named file already exists (this then prevents already-generated traces from being overwritten).

All the elements of the array storing the traces that have been written in are then written out to the file as a set of 32-bit integers (as to why these are all 32-bit integers being written, see section 4.5.4). Once this has been done, the file stream buffer is flushed (with the `CommitL()` function), and the file and file server are closed.

Application startup code

In the application's `NewApplication()` function (which gets called when a new application is created), a new ChunkHeap (see section 3.3) that uses the tracing allocator needs to be created, and this heap needs to be set to being the active heap. This is done with the following code:

```
TInt heapCount=0;
TName n;
n.Format(_L("TESTHEAP%d"),heapCount++);
MyHeap* mh=(MyHeap*)User::ChunkHeap(&n,0x1000,0x100000);
User::SwitchHeap(mh);
```

The `MyHeap` header file also needs to be included at the start of the file containing the `NewApplication()` routine.

After the modification, the application needs to be compiled and then run through the emulator.

4.3 Processing trace files

Having obtained the trace files, they then need to be post-processed to be able to obtain meaningful results. This was done by writing a C program

to read in the trace files (the code itself contains the same `struct` as the `MyHeap` tracing `struct`), and then output a set of numbers which could then be fed into some graphing software to generate interesting graphs.

The code also kept a running total of the memory allocated (to produce a "time in bytes"), the amount of live memory used, the total amount of memory that would have been used given no internal fragmentation (i.e. if the memory really allocated was the same as the memory requested), the number of objects allocated of a particular size, the maximum heap size, with the live memory usage at that point, the maximum amount of live memory (these were to be able to get a measure of fragmentation comparable to [8]; see section 2.4.2), and a running total of the internal fragmentation (this was taken as the ratio of the bytes allocated to the bytes requested), which was then divided by the number of allocation operations to obtain an idea as to an average amount of internal fragmentation.

To keep track of the locations where allocations were taking place, so they could be matched with their conjugate free operations, a linked list containing addresses and sizes was used[4]. Each time an allocation was recorded, a new element was added to the start of the linked list, containing the address and size of the object allocated. This was then added to the running total of live memory used and total memory allocated. For a free operation, the element of the linked list containing that particular address was found and removed from the list; the size recorded was then subtracted from the total live memory. During a reallocation operation (i.e. the cell had not moved), the element of the linked list containing the particular address was found, and the size recorded was updated, along with the live memory; if the new size was larger than the old size, then the running total was incremented by the difference in sizes.

To keep track of the number of objects allocated of a particular size, an array of integers was set up for objects ranging between a certain range of sizes (defined by the macros `MINSIZE` and `MAXSIZE` at the start of the source code). Each time an object of a particular size was allocated, the corresponding location in the array was incremented. For any objects smaller than `MINSIZE` or larger than `MAXSIZE`, a linked list was used, each element containing the size of object and number of objects stored. If an object with a new size had been allocated, then a new entry was created in the list, making sure that the object sizes were kept in ascending numerical order as the list was chained down.

When compiled, the executable took two command-line parameters (see

[4]If this was implemented as a hash table it would have been faster to look up; however it would have added extra complexity to the code

Appendix B) – the first a single character denoting the type of output required ('human-readable', memory sizes, or object sizes), and the second the name of the file containing the traces.

With all the outputs, the program printed out the size of a free-list cell, the minimum allocatable cell size, the two measures of fragmentation used by [8] and a measure of internal fragmentation, all written to `stderr` (the Unix standard error stream; by default the console).

With 'human-readable' output, the program printed out a set of lines corresponding to the actual traces on `stdout`; the code being used to do this follows (`trace` is the name of the `struct` that contains each memory trace).

```
printf("Op: %c\n",trace.op);
printf("Requested: %d\n",trace.request);
printf("Really Allocated: %d\n",trace.alloced);
printf("Address: %lx\n",trace.address);
printf("Start tick: %ld\n",trace.starttick);
printf("End tick: %ld\n",trace.endtick);
printf("Heap size: %ld\n",trace.heapsize);
```

When an output of memory sizes was requested, the program printed out a tabulated list of numbers on `stdout`, corresponding (in order) to the total amount of memory allocated, the total live memory, the total memory that would have been live given zero internal fragmentation, and the heapsize at each particular operation. This output could then be redirected to a file, and used to generate some graphs of heap size and live memory usage as allocations took place.

When an output of object sizes was requested, the program printed out a tabulated list of numbers on `stdout`, corresponding to object sizes and number of each size allocated. This output could then be redirected to a file, and used to generate a graph of number of objects vs. object size.

To generate the graphs of the memory allocator characteristics, the outputs from the trace processor were used as the input to `gnuplot`, a free graphing program available on most Unix systems. The commands that needed to be given to `gnuplot` to generate the graphs are given in the User Manual (Appendix B).

4.4 Test and Debug

4.4.1 Statement of test plan

An initial test of the code was to make sure it compiled properly, and that some trace files could be generated without any errors occurring.

Once some traces were obtained, to make sure that they appeared to be written correctly (before they were processed), the files were inspected by eye using a hexadecimal dump (in this case, the files were copied to a Unix machine, a read using the utility `hexdump(1)`). Because each trace file contained at least a few hundred operations, the files could not be examined too closely in this way, but it gave a good indication that the traces were written out correctly.

4.4.2 Summary of test results

Although the code itself had compiled properly, using the code from within an application startup caused it to fail. This is discussed in section 4.5.1.

Once the traces had been obtained, an initial inspection found that they had been created successfully. However, some issues were then seen to arise when the trace files were copied to another machine. This is discussed in section 4.5.6.

4.5 Issues

4.5.1 Application startup

Unfortunately, having compiled the `MyHeap` code, and a test application[5] which switched from the Symbian allocator to the tracing allocator in its `NewApplication()` routine (see section 4.2.3), the test application was then found to fail on startup. Investigation into this problem found that it was failing because some of the control environment base class objects had already been allocated before the heap was switched, and subsequently were attempting to access the old heap, which was prevented by the OS.

After some deliberation and discussion with Symbian, it was decided to modify some of the code in the OS which launches applications, to enable the heap (and memory allocator) to be used before the application itself starts up.

4.5.2 EIKDLL

The operating system application launching code that needs to be modified is located in `src\COMMON\GENERIC\app-framework\uikon\coresrc\EIKDLL.CPP`

[5] An application that displayed "Hello World" on the screen, which is part of the Symbian development kit as sample code, was used, as this was one of the simplest applications available, and hence easy to experiment with

in the function called `AppThreadStartFunction`. The code that was used to modify this file and the method of compilation is given in the System Manual in section A.2.

Having done this, switching to the tracing allocator worked properly, and applications were able to run and output traces.

4.5.3 CodeWarrior bug

When attempting to compile the application launcher code, a bug was encountered in CodeWarrior whereby long path names were not handled correctly, and the build failed. This necessitated a build from the command line (this is described in the System Manual, section A.2).

4.5.4 Struct alignment issues

Initially when writing out the traces (see section 4.2.3), the operation was treated as a 16-bit integer, with the rest of the information being 32-bit integers. However, although this was then written out correctly by the `DumpTrace()` function, when processing the trace files (see section 4.3) the C `struct` was aligned so that each 32-bit integer fell on a 4-byte boundary. This then meant that when the trace was read in, the two bytes of the operation were read in correctly, but then the next two bytes were ignored (the two bytes being part of the next item in the `struct`: the amount of memory requested) so the next item, when reading in, was aligned to a 4-byte boundary. To get around this issue, every item when writing out and reading in the traces was treated as a 32-bit integer.

4.5.5 Timing issues

Another encountered issue involved obtaining the timing information – due to the allocation and free routines running quickly, it would be necessary to obtain microsecond accuracy to be able to tell how long it took for the routines to run. From the Symbian documentation, it appeared that the `gettimeofday()` call (from their implementation of the C standard library) looked promising – one of the returned values is the system time in microseconds. However, having used this when obtaining traces, it turned out that this call was only updated every 5 milliseconds. Having spoken with Symbian about this, it turned out that user processes are not given access to any timer finer than a few milliseconds. There exists a `Kern::TickCount()` method which should return more accurate timing information; however, this is only accessible from a kernel-space process, and causes an error if attempting to

call it from a user-space process. It was therefore decided to not attempt any further to obtain more accurate timing information, due to time constraints, and leave this for further work (see section 7.2).

4.5.6 Endian issues

As a result of checking the trace files by inspection, it was found that when copying the trace files to a computer that used a different architecture (the trace files were generated on a computer with an x86 architecture, and then copied onto computers using SPARC or PowerPC processors), the bytes in the trace file were swapped. This is due to endian issues (the order in which bytes of words are held in memory) – the x86 is little-endian (storing least significant byte first), whereas the SPARC and PowerPCs are big-endian (storing the most significant byte first). To work around this issue, a short program (listed at the end of the report with the rest of the code) was written to do the appropriate byte shifting so the traces could be read on a big-endian architecture. This read in the name of a tracefile on the command line, and wrote out a byte-swapped version to a file of the same name, but with `.end` appended.

Chapter 5

Symbian OS allocator results

The method for obtaining the results has already been discussion in section 4.3. This chapter shall discuss some interesting aspects of the processed results. All the graphs obtained from the results are included in the Appendix, before the code section.

When processing the results, it was found that in certain places, the heap size appeared to suddenly reduce back to its initial size while there was still a large amount of allocated memory. Further investigation (using the "human-readable" output option of the trace file processor) found that the objects being allocated when the heap size went to its initial size were allocated from a different memory location, leading to the conclusion that the applications, during the course of their execution, created multiple heaps.

It was therefore decided to concentrate the investigation of the allocator external fragmentation performance on one heap only: the initial heap to be used. Another program was written (whose code can be seen at the end of the report) which ran through a given tracefile, and produced another tracefile as an output, containing only allocations from the initial heap. Due to the maximum size of a heap being set at 16MB, or 1,000,000 in hexadecimal, (this is one of the parameters to the ChunkHeap method call when creating a new heap, given in the System Manual (Appendix A)) the top 8 bits were used as a mask on the address (the bottom 24 bits are the parts of the address that may vary in a heap) to see if a particular operation took place on the initial heap.

5.1 Characteristics of the Symbian OS allocator

All the graphs obtained from investigating the initial heap appear to be very similar; this is because as an application loads, a new (graphical) environment needs to be set up for it: the traces plotted on the graphs are of the memory usage and heap size as this environment is being set up. One of the graphs is shown in figure 5.1.

The external fragmentation of all these heaps were identical; using a measure of fragmentation comparing the amount of live memory to the heap size at the point where the heap size is maximal gives a fragmentation measure of 7.0%. Using a measure that compares the maximum heap size to the maximum amount of live memory gives a fragmentation measure of 6.9%. The details of these fragmentation measurements are given in section 2.4.2. The first measure corresponds quite well to the average fragmentation for a first-fit address-ordered allocator obtained by [8]: this measured 6.63% average fragmentation. However, using the second measure, [8] found the average fragmentation to be 2.30%. The discrepancy between the two values occurs because the allocations obtained from the Symbian OS are quite different to the allocations in the software that was used in the Johnstone and Wilson study – in the Symbian allocations, the heap size followed the live memory usage fairly closely.

To obtain a measure of the internal fragmentation, the entire trace file was processed (the internal fragmentation is not dependent on the heap size; therefore having multiple heaps does not matter), and an average value for the percentage fragmentation obtained; the percentage for each value was worked out using $\%frag = 100 * \left(\frac{allocated\ memory}{requested\ memory} - 1\right)$. The results of this are given in table 5.1.

The large measures of internal fragmentation show that rhere is a large amount of memory wasted; a lot of this is likely to be due to using a first-fit allocator (and not being able to split off blocks), instead of a best-fit.

5.2 Characteristics of smartphone apps used for the traces

To provide a better overview of the frequency of object allocation when running an application, the allocated objects across the entire run of the program (i.e. across all the heaps) were considered.

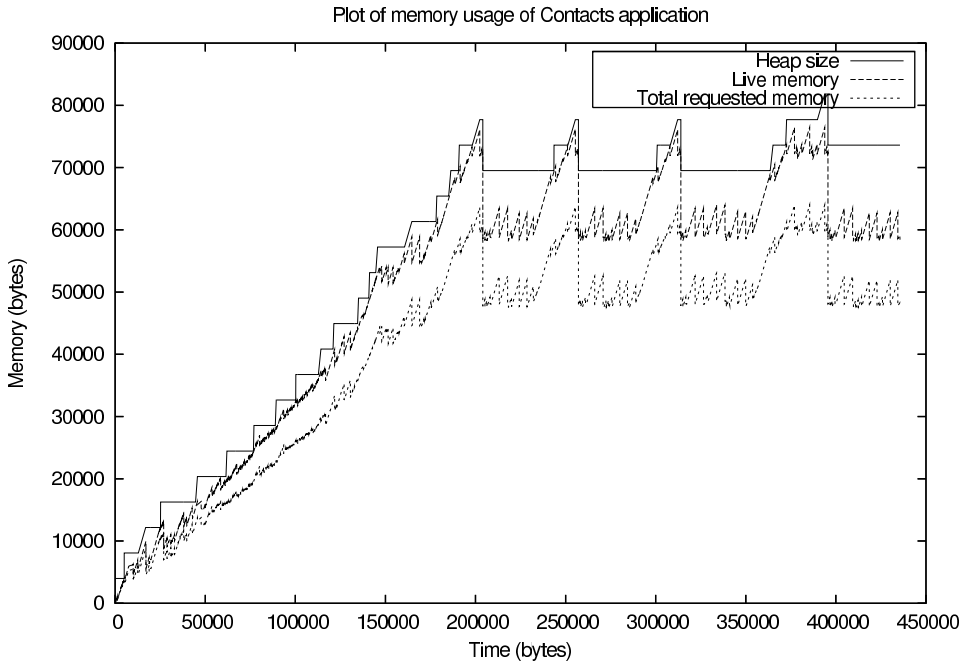

Figure 5.1: Memory usage in the Contacts application

Application	Internal fragmentation (%)
Agenda (1st run)	63.0%
Agenda (2nd run)	65.0%
Contacts	66.0%
HelloWorld	66.0%
Messaging	66.0%
Message editor	66.0%
Time (alarm)	66.0%

Table 5.1: Internal fragmentation of the test applications, across the entire run of the application

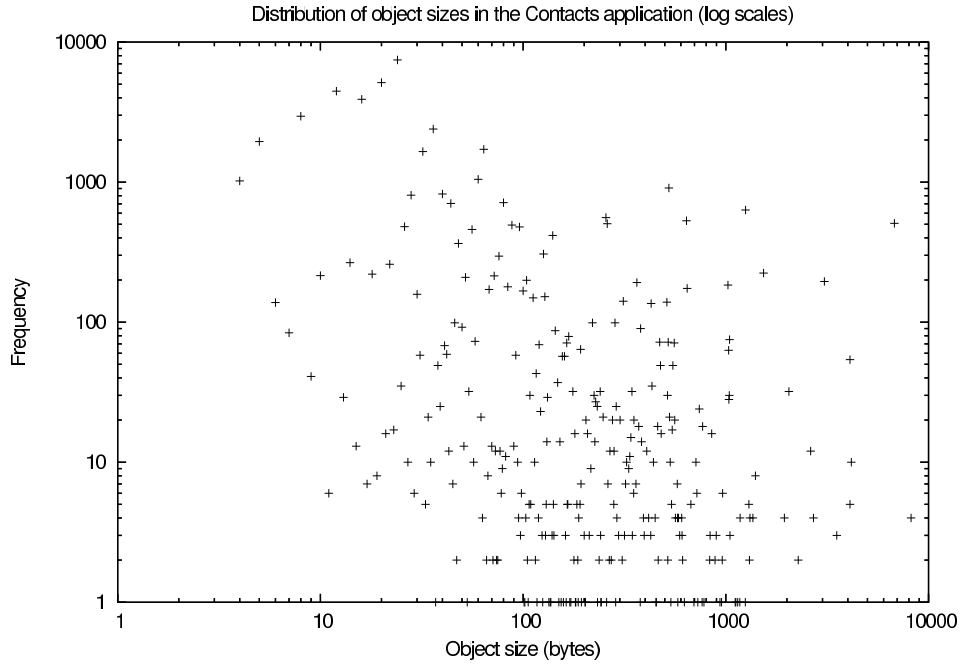

Figure 5.2: Distribution of object sizes in the Contacts application

One of the graphs obtained is seen in figure 5.2. As can be seen, there is a fairly random spread to the objects allocated; the size of the majority varies between 10 bytes and 1000 bytes. All of the other applications resulted in similar looking graphs (hence only one is produced here; the others are found in the Appendix).

5.3 Overheads

Each free and allocated block has a header associated with it, which contains such information as the length of the block, and possibly (if the block is free) a pointer to the next block on the free list. This necessarily adds some (unavoidable) overhead to the memory allocator, and hence also requires a minimum allocatable block size (guaranteeing that there is enough space for all required headers). In the Symbian allocator, this is set to 12 bytes, so any requests for smaller blocks get rounded up to 12 bytes.

From examining the Symbian code, an allocated block only has 4 bytes of header, which contains the size of the block. Therefore each allocation will use an extra 4 bytes over the memory that is said to be live. This does not actually use a great deal of extra memory – if there are 5,000 live blocks

of data in use at any one time, then there is an overhead of approximately 20kB.

Chapter 6

Critical assessment of Symbian OS framework

6.1 Design of a segregated free list allocator

Given that the tracing extensions to the Symbian allocator worked, and the system had been set up to use this tracing allocator, it was then decided to attempt to implement a segregated free list allocator to see how well the Symbian OS infrastructure coped with using a replacement allocator. In theory, this should just be a case of overriding some methods in the Symbian allocator, and switching to the new allocator, all of which is supported by the framework (see section 3.4). As such, it would be useful to enhance the `MyHeap` code (see section 4.2.3) with the ability to use the segregated free list allocator.

In an ideal segregated free list, every possible allocatable size will have its own set of free lists (see section 2.2.2). However, this requires having up to 2^{32} free lists (on a 32-bit processor), and hence is not practical, so a set of free lists corresponding only to a limited range of sizes is more useful. An appropriate range of sizes can only really be found by empirical observation to find out how big allocated objects tend to be. If the set of free lists only corresponds to a limited range of sizes, then consideration must be given to any cell that is to be allocated outside the chosen range of sizes. To address this, two extra free lists, a so-called 'small' list and 'large' list, are used; the small list holding a free list of blocks smaller than the minimum size in the segregated list, and the large list holding a free list of blocks larger than the maximum. These lists can then be treated as sequential free lists, and can be allocated from using a sequential fit algorithm (this often tends to be first-fit).

Additionally, it would be useful for this segregated free list allocator to be used as a base for an exact-fit peer group memory allocator as described by Chris Clack (see section 7.2); where this has been done (hence causing some differences between this implementation and a "standard" segregated free list implementation) is noted in section 6.2.2.

6.2 Implementation

6.2.1 Overview

To be able to extend the `MyHeap` code with the segregated free list allocator, it was decided to make use of the C pre-processor to select the allocator to use (i.e. use of the Symbian allocator, or the segregated free list allocator). This would then allow for easy switching between the allocators – only requiring uncommenting one line of code, commenting out another line, and recompiling. Hence at the start of the header file, `myheap.h`, the following definitions are set up:

```
//#define segregatedFreelist // Segregated free list with linear
                             // searching
#define symbianMalloc    // Use the original Symbian allocator

// Define, if we want tracing
#define tracing
```

As the code above stands, the original Symbian allocator, with tracing extensions, is to be used.

The entire code can be seen at the end of the report; only important aspects of the implementation of the segregated free list allocator will be discussed. As to how the definitions given above are used, this is best illustrated when looking at the code in its entirety.

The operation of the allocator is intended to mimic the operation of the Symbian allocator as closely as possible (see section 3.3), merely using a different data structure to hold the free blocks. Hence discussion of the routines in the code will concentrate on the differences between this and the Symbian allocator. Operations not discussed should be assumed to be identical to the Symbian implementation, discussed in section 3.3.

6.2.2 Structure of the free lists and blocks

Free block

A block on the free list contains four items in the header:

1. Its length and availability
2. The length and availability of the previous block in the heap
3. A pointer to the next free block in the free lists
4. A pointer to the previous free block in the free lists

An allocated block only contains the first two items in its header.

The availability is marked with the top bit – if it is set, then the block is free; if it is cleared (i.e. 0), then the block is in use. This may be regarded as non-standard in an implementation of a linked list; it is implemented in this way to provide a basis for an extension of this code to implement Chris Clack's allocator. Using the length and availability of the previous block in the heap provides an easy mechanism for coalescing without having to search through the entire heap for the block next to one being freed.

However, if the length of a block is requested, then the top bit must be ignored (the top bit is usually designated as a sign bit – if it is set, then the number being represented is negative). Hence the header file contains some useful macros such as (amongst others):

```
#define GETLENGTH(s)        ((s)->len & 0x7FFFFFFF)
#define ISFREE(s)           ((s)->len < 0)
```

Using these macros, if the implementation changes, only the macro needs to be changed instead of finding every location in the code: the code itself should continue working.

Free lists

All the free blocks are held in a doubly-linked list. In addition to this, the segregated lists are implemented by holding a list of a particular size as elements in an array; the list holding free blocks of size `MINSIZE` (which is defined in the header file) is held in position 1 in the array; the rest of the array holds free lists of increasing size, up to `MAXSIZE` (also defined in the header file), which is held in the second-last element. The header file may contain, for example, the following definitions:

```
#define MINSIZE 128
#define MAXSIZE 256
#define ARRAYTOP 2+MAXSIZE-MINSIZE
```

When protected variables are declared as part of the `MyHeap` class definition, the array is held as an instance variable of a heap:

```
SCell* freelists[3+MAXSIZE-MINSIZE];
```

An `SCell` is the name given to a free block.

Because the segregated list only holds free lists between the values of `MINSIZE` and `MAXSIZE`, allocations smaller than `MINSIZE` or larger than `MAXSIZE` need to be catered for – this is done by having a 'small block' free list, and a 'large block' free list, which are held in the segregated list array at position 0 and the last element respectively. Both of these lists act as a first-fit sequential allocator.

When a block is added to the free lists, it is placed at the head of its particular size free list (see section 6.2.9 for more details), and then tied in with the doubly-linked free list. Removal of a block from the free lists also happens from the head of its free list (see section 6.2.8 for more details); hence each individual free list (as referenced from the segregated list array) contains blocks that are held in LIFO (Last In First Out) order.

6.2.3 Allocation

When a block is to be allocated from the free lists, a call to the function `getCell()` is made (this is described below in section 6.2.7) – this returns a block of at least the requested size, or `NULL` if a block was not able to be found.

The block is then marked as being in use, and checked to see if it can be split to hold a new block from the excess memory between the requested size and the size of the returned block. If not, then, assuming the returned block is not the last block in the heap, the next block in the heap is obtained, and its previous block size element is set to being in use. The block is then removed from the free lists (see section 6.2.8), marked as being in use, and returned.

If there is enough space to create a new block, then a new one is created, marked as being free, and its previous block size set to the size of the requested block size, and marked as being in use. If this new block is not at the top of the heap, then the next block in the heap is obtained, and its previous block size element is set to the length of the newly split off block, and marked as being free. If the new block is at the top of the heap, then the

variable `iTopMostCell` is updated with the address of the new block. The old block is then removed from the free lists, the newly split off block added (see section 6.2.9), the size of the old block is then updated to match the requested memory size, marked in use, and the block is returned.

6.2.4 Freeing

When a block of memory is freed (i.e. is being returned to the free lists), it is initially marked as being free. First an attempt is made to coalesce the block with the one below it in the heap. If this is possible (i.e. the block below is free), then that block is removed from the free lists, and its size increased by the size of the newly freed block. The pointer to the newly freed block is then changed to point to the start of the previous (now extended) block.

If the block is not at the top of the heap, then the next block in the heap is obtained, and its previous block size element is updated to reflect the size of the new block (in case it had been coalesced above), and the previous block element is marked as being free. An attempt is made to coalesce forwards – if this is possible, then the size of the new block is extended by the size of the next block and the next block removed from the free lists; if this is then the last block in the heap, the variable `iTopMostCell` is updated to point to the start of the new block; if it is not the last block in the heap, then the previous block element of the next block in the heap is updated to reflect the new block size.

Once any coalescing has been attempted, then an attempt may be made to shrink the heap – this follows the same criteria as given in section 3.3.2, as to whether or not an attempt is made.

Finally, the newly freed block is added to the free lists (see section 6.2.9).

6.2.5 Reallocation

If a block is to be extended, then a new block is allocated, the contents of the memory copied from the old block, and the old block is freed up. For simplicity, and to assist with the debugging process, an attempt is currently *not* made to attempt to grow a block in place (however, there is the framework of a procedure in place, which needs further development before it can be used).

If a block is to be shrunk, and there is enough space left over to be able to form a new block, then a new block is created from the excess space, and marked as being free. The value of the previous size element of the new block is set to the requested reallocation size (the new length of the shrunk block). This new cell is then added to the free lists.

6.2.6 Growing the heap

The top-most cell (value of the variable `iTopMostCell`) is obtained, and a check is made whether it is free or not. If it is, then the size extension requested is reduced by the size of that cell:

```
SCell* pT=iTopMostCell;
TBool top_is_free = ISFREE(pT);
TInt extra=top_is_free ? aSize-(GETLENGTH(pT)) : aSize;
```

If the size of the extended heap is not greater than the maximum heap size, a request is made to the kernel to allocate more memory. If this is successful and the top-most block is free, it is extended; if the top-most block is not free, then a new block is created and added to the free lists.

6.2.7 Obtaining a block

The procedure for obtaining a block returns a block of at least the requested size. Initially, the requested block size's location in the array of free lists is obtained via the function `ListLocation`, defined as:

```
inline TInt MyHeap::ListLocation(TUint aSize) const
     /**
      * Returns location in the segmented free list
      */
{return (aSize < MINSIZE ? 0 : (aSize > MAXSIZE ? (ARRAYTOP) :
                                  (1+aSize-MINSIZE)));}
```

The `inline` keyword asks the compiler to insert the generated code as part of the calling function, instead of creating an entirely new function (with associated jumps in the executable code). If the requested size (`aSize`) is smaller than `MINSIZE`, the value 0 is returned (which corresponds to the bottom element of the array of free lists); if the requested size is larger than `MAXSIZE`, then the value `ARRAYTOP` is returned, which corresponds to the top element of the free list array. Otherwise, the value corresponding to the array element of the free list of the requested size is returned (this starts at 1, because the 0th element is taken by the small block list).

Once this has been obtained, then a search is made up the free lists to find the first non-empty location that is at least as large as the requested size (which may well be the same as the location returned by the `ListLocation` function). If this is not found, then a `NULL` value is returned. If the first non-empty location is in the small list, then a first-fit match is attempted; if this is unsuccessful, then the list location is set to the first segregated free

list, and a linear search is performed for the next available block (up until the large list location is hit). If the first-fit match is successful, then the block is returned.

If the first non-empty location is one of the segregated lists, then the block at the head of the list is returned.

If the large list contains the first non-empty location, then a first-fit match is attempted; if this is unsuccessful, a `NULL` pointer is returned, as there is no block of at least the requested size that is free. If the match is successful, then that block is returned.

6.2.8 Removing a block from the free lists

When a block is to be removed, the previous and next pointers on either side of the doubly linked list (if they point to valid blocks) are set up to point to each other:

```
SCell* pP=aCell->prev;
SCell* pN=aCell->next;
TInt aSize=GETLENGTH(aCell);
TInt listlocation=ListLocation(aSize);
if (pP)
{
    pP->next = pN;
}
if (pN)
{
    pN->prev = pP;
}
```

`aCell` is the block to be removed from the list

If the block that is being removed is at the head of its linked list, then care must be taken to update the array holding the linked lists – if there is another block of the same size following, the element of the array needs to be updated to point to this block. If there is not a block of the same size following (or, in the case of the small list, a block not smaller than `MINSIZE`), then the element of the array needs to be set to `NULL` (i.e. marked as being empty).

6.2.9 Adding a block to the free lists

When adding a block to the free lists, initially a check is made to see if a block of the same size already exists (i.e. the element of the free list array is

not `NULL`). If such a block does exist, then the new block is added at the head of the list (with the previous and next pointers being set up appropriately).

If such a block does not exist, then it is necessary to find the next larger block – this is done with a call to `getCell()` (described in section 6.2.7). If a block is returned, then the previous pointer of the newly added block is set to the same as the previous pointer of the next larger block, the rest of the pointers updated appropriately, and the new block becoming the head of a new list.

If a block is not returned (i.e. the added block is the largest in the heap), then the next smaller block needs to be found, which is done by a call to the function `getSmallerCell()`. The value returned from this function is used as the value of the previous pointer in the newly added block, the new block's next pointer is set to `NULL` and other pointers are updated appropriately, at which point the new block becomes the head of a new list.

The `getSmallerCell()` function works very similarly to the `getCell()` function described in section 6.2.7, except that it returns a cell that is the next smaller in size than the requested size (or `NULL` if such a block does not exist); hence searching is performed down the free list array, instead of up the array.

6.3 Test and Debug

Testing of the allocator (and hence the Symbian OS allocator extensibility framework), consisted of attempting to compile the code and run various applications.

In the course of this, a number of issues with the code were found and fixed until the shell application (the basic user interface) was able to run.

6.4 Issues

Even though this allocator was implemented to mimic the behaviour of the Symbian allocator, and applications were run successfully using the tracing extensions (so successfully using `MyHeap`), when running applications with this allocator, failures were observed in varoius places.

One of the observed failures was an error marked as `CONE:8` when shutting down an application. This, according to the Symbian documentation, is an error that occurs when the OS environment found window server resources had not been freed. It was not possible (mainly due to time constraints – tracking down the problem would have taken many days) to find out which

resources in particular had not been freed. It was, however, interesting to note that this error occurred when running the replacement allocator, but not with the Symbian allocator (which may imply some dependency on the Symbian allocator).

Another observed failure (either when starting some applications, or during the course of running the applications) resulted in CodeWarrior stopping execution of the emulator and returning to the IDE. Running the emulator outside of the CodeWarrior environment showed that the execution was terminated when an error marked as `KERN-EXEC:3` occurred. According to the Symbian documentation, this error is thrown when an unhandled exception occurs; it is most common when an access violation occurs, for example by derefencing `NULL`. The cause of this error was not able to be tracked down, mainly due to CodeWarrior not launching a debugger session (which may have helped track the cause of the error), but terminating execution.

Chapter 7

Summary and Conclusions

This chapter contains a brief recap of the work carried out in this report. Further work is suggested to continue the investigation, and the report is concluded with some personal comments about the work.

7.1 Summary

An investigation into the fragmentation performance of the Symbian Operating System memory allocator and an assessment of the extensibility of the allocator framework has been presented.

The Symbian allocator implementation is a first-fit address ordered allocator which uses coalescing. Traces of allocator activity had been obtained to get a measure of fragmentation performance, and this lead to the observation that each application created multiple heaps during the course of its execution. It was therefore decided to concentrate the measures of fragmentation on only one heap (the first heap created during execution). Two measures of external fragmentation and a measure of internal fragmentation were obtained (described in section 4.3).

The external fragmentation performance was found to be good, with only about 7% fragmentation. However, internal fragmentation performance was found to be poor, with fragmentation measures between 63 and 66%.

Additionally to the allocator performance, some statistics on the frequency of object allocations were obtained which may be useful to note when implementing a memory manager.

An investigation into the extensibility of the allocator proved interesting – some limitations became apparent, such as the failure to switch to a new heap from within an application: it was necessary to modify the application

launching code in the Operating System. Further investigation into the extensibility by implementing a segregated free list allocator lead to an apparent weakness in the environment: the fact that window server resources were not freed properly when running the replacement allocator but not the Symbian allocator implies some extra dependence on the Symbian implementation.

Another weakness turned out to be the development environment itself – although CodeWarrior does include a debugger (which at times proved to be very useful) there were also times where the debugger did not launch, merely terminating execution of the emulator without giving any indication as to why the emulator was terminated. The emulator itself had to be run outside the CodeWarrior environment to find out the error that occurred, causing it to be terminated.

7.2 Further work

There are a number of ways that this investigation can be enhanced and extended; indeed the study itself discovered issues that had to be left for further study. Ideas and suggestions for further work are listed below.

One enhancement is to modify the trace processing program to be able to deal with multiple heaps from within one trace file. Alternatively, the `MyHeap` code could be modified so that instead of keeping the array of `struct`s (the traces) as a global variable, each heap gets its own local array, which is then written out when the heap gets destroyed: this would involve creating a destructor method.

There may be some code issues with the segregated free list implementation (causing the `KERN-EXEC:3` panics) that remain unresolved due to time constraints: it may be possible to find these after careful inspection of the code, or after time spent working with the debugger to find problems. Some debugging code remains with this implementation, which may be useful as a base to work with.

Still with the segregated free list, it would be useful to implement the function that attempts to grow a reallocated cell in place – the framework exists in the code, but the functionality was not implemented due to trying to keep the code as simple as possible for the purposes of testing and debugging.

Some of the design decisions for the segregated free list allocator were taken so it would be possible to extend the code further by implementing an allocator described by Chris Clack[1], without taking a great deal of effort –

[1]This is described in UK Patent Application 9924061.1: A Data Structure, Memory Allocator and Memory Management System, Example of Technique

the main modification needed is to use a non-linear method of searching the free lists (a description of the allocator is outside the scope of this project).

One of the issues that was encountered fairly early on was that of obtaining timing information for the memory allocator – i.e. measuring how many microseconds it took to perform an allocation or free operation. However, due to a lack of access to sufficiently accurate timing information, it was decided not to pursue the timing information further for this project. However, the Symbian OS kernel does have access to more accurate timing information, but this is not accessible to a user-space process. It may be possible to develop a simple device driver, accessible from user-space processes, which provides access to the kernel timer devices. Doing this would then open up more ways of assessing the Symbian allocator and comparing it to other implementations (comparing speed of allocations).

7.3 Conclusions

Overall, I am happy with the outcome of the study. Although a number of issues were enountered, they had been overcome sufficiently to generate some useful results.

The Symbian OS allocator extensibility framework still has issues that need to be worked out before it can be used properly in a non-Symbian development environment (e.g. third-party software authors who write software for the Symbian platform); one of the major issues is (currently) the lack of documentation for supporting this framework. Hopefully this report will provide some indicators as to where improvements may be made.

The Symbian allocator performance with regards to external fragmentation is in line with the fragmentation performance of a first-fit address-ordered allocator, as documented by Johnstone and Wilson[8] – their study showed an average external fragmentation of 6.63% when comparing maximum heap size to the amount of live memory at that point, whereas the investigations here gave the fragmentation at 7%. Internal fragmentation was shown to be quite high (around 65%), which indicates that there is a large amount of unusable memory. Although this is normally not likely to be much of an issue for a device with a large amount of memory (such as a workstation), it can become a large problem on a low-memory device such as a smartphone; for example allowing fewer applications to run concurrently, or limiting the amount of data that can be stored.

Internal fragmentation is a phenomenon that appears to be investigated less often than external fragmentation; nevertheless it is extremely worthy of more investigation, such as getting statistics for the performance of various

allocator algorithms. It stands to reason that a best-fit allocator (or an allocator that uses a best-fit policy, such as a segregated free list allocator) would provide the best internal fragmentation performance, as it attempts to find a block closest in size to the requested size, instead of just taking the first block that is large enough.

To conclude, I would say I have produced a good solid study. The objectives of the project were met; additionally I believe that this is the first study that has obtained memory allocation traces from smartphone applications. If this project proves useful for Symbian in further developing their allocator and framework, or if it provides a useful base for anyone extending the work described, I would consider it a great success.

Appendix A

System manual

This manual outlines the main technical details that would enable a reader to set up an environment that allows for further investigation of the Symbian memory allocator and allocator extensions.

A.1 Software Installation

The installation of the software follows a number of steps:

1. Metrowerks CodeWarrior for Symbian v. 2.8 needs to be installed. This is available as a trial edition for download from `http://www.metrowerks.com`[1], and acts as a standard Windows installer package.

2. The Symbian emulator needs to be installed – this is available as a development kit (DevKit) from Symbian, and contains the emulator as well as some source code. Again, this acts as a standard Windows installer package.

3. Before the emulator can be run, a default device needs to be set up. Running the command `devices` from the command line will give a list of devices (emulator targets) that can be used. The TechView device needs to be set as the default – to do this, run `devices -setdefault @03282a_Symbian_OS_v8.0b:com.symbian.TechView` from the command line.

4. Once the default device has been set up, running `epoc` from the command line should launch the emulator.

[1] If working on this at Symbian, a full edition of this version is available on the Symbian network

5. Copy the `MyHeap` code from the CD supplied[2] into a directory on the hard disk.

A.2 Code Compilation

Once the software has been installed, it is then necessary to set the system up to be able to use the tracing allocator (or other allocator implementations). This also involves a number of steps:

1. Go to the directory containing the `MyHeap` code and edit `myheap.h` and check that the line

   ```
   #define symbianMalloc
   ```

   is uncommented, and the other allocator selections are commented out.

2. At the command prompt in that directory, enter the following commands to build the system:

   ```
   bldmake bldfiles
   abld build winscw
   abld freeze winscw
   abld build winscw
   ```

3. Go to the directory `src\COMMON\GENERIC\app-framework\uikon\coresrc` that is located under the Symbian DevKit installation directory (normally under C:\Symbian), and open the file `EIKDLL.CPP` for editing.

4. Find the function named `AppThreadStartFunction`, and add the following code before the line `if(commandLineHBuf == NULL)`:

   ```
   const TUint KMaxFileLength=128;
   const TBufC<KMaxFileLength> cached(
                 *STATIC_CAST(const TDesC*, aParam));
   GlobalFreePtr(aParam);

   // Set up a new heap here
   ```

[2]This is also available as a `.zip` file from `http://www.eco.li/writings/msccs/myheap.zip`

```
MyHeap* mh=MyHeap::ChunkHeap(NULL,0x1000,0x1000000);
RHeap* oldheap=User::SwitchHeap(mh);

HBufC* const commandLineHBuf=cached.AllocL();
```

5. After the line `EikDll::RunAppInsideThread(cmdLine);`, add the following code:

   ```
   MyHeap::DumpTrace();
   User::SwitchHeap(oldHeap);
   ```

6. Save the file, and open the file `eikcore_base.mmh` in the same directory.

7. Add `MYHEAP.LIB` to the list of libraries (at the end of one of the lines which starts with `library`, and save the file.

8. On the command line, go to the directory `src\COMMON\GENERIC\app-framework\uikon\group` that is located under the Symbian DevKit installation directory (normally under C:\Symbian).

9. Build the library by entering the following commands:[3]

   ```
   bldmake bldfiles
   abld build winscw
   ```

A.3 CodeWarrior use

Once the environment is set up, the `MyHeap` project can be imported into CodeWarrior. This is done by selecting `File | Import project from .mmp file` from within the IDE, selecting the appropriate SDK and then selecting the appropriate `.mmp` file. Compilation is achieved by pressing F7 or selecting `Project | Make`, and the emulator can be launched with the debugger by pressing F5 or selecting `Project | Debug`.

Tracing may be turned on and off by uncommenting or commenting out the line `#define tracing` in the `MyHeap.h` file, and the implementation of the segregated free list allocator (see section 6.2) may be used by uncommenting out the line `#define segregatedFreelist` and commenting out the line `#define symbianMalloc`. These necessitate a recompilation of `MyHeap`.

[3] If this build is attempted from within the CodeWarrior IDE it will fail, due to a bug in CodeWarrior not handling long path names properly; hence this step *must* be done at the command line

Note that the compiler may generate an error about 'Exports not yet frozen'. Unfortunately, CodeWarrior does not have the ability to freeze the exported functions (see section 3.2), so, if this occurs, the project must be recompiled on the command line. This is achieved with the following commands:

```
abld reallyclean winscw
del myheapwins.def
abld build winscw
abld freeze winscw
abld build winscw
```

Included with the `MyHeap` code is a batch file called `rebuild.bat` which automatically runs the above commands.

Appendix B

User Manual

Once the system has been set up (with tracing enabled) and run as described in the System Manual, some traces should be obtained by running some of the programs available with the emulator. This User Manual deals with the processing of the trace files once they have been written out. Please note that the processing is assumed to be performed on a Unix system with compilers available; if this is not the case, it is recommended to install Cygwin, a free Unix-like environment for Windows, available from `http://www.cygwin.com`. The installation of this environment is beyond the scope of this document.

The trace files themselves will be found in the emulator's `C:\traces` directory; this maps (assuming default installation paths) to the directory `C:\Symbian\03282a_Symbian_OS_v8.0b\bin\techview\epoc32\winscw\c\traces` on the host system. Once created, these files should be moved to another directory.

B.1 Processing the trace files

The file `processtracefile.c`, included with the `MyHeap` code, is used to process the files. Initially, it needs to be compiled. This is done at a Unix shell prompt with the command `cc -o processtracefile processtracefile.c` which will generate an executable called `processtracefile`. If this is run with no arguments, then the following output is displayed:

```
Wrong number of arguments.
Usage: ./processtracefile [hmo] tracefile
h: Produce human-readable output
m: Produce a dump of memory sizes
o: Produce a dump of object sizes
```

With all the outputs, the size of a free-list cell, the minimum allocatable cell size, two measures of external fragmentation used by [8] and a measure of internal fragmentation all written to `stderr`, which is the Unix standard error stream (by default this goes to the console).

When "human-readable" output is specified, the program prints out a set of lines on `stdout` (the Unix standard output stream, by default this is the console) that correspond to the traces, for example:

```
Op: A
Requested: 70
Really Allocated: 84
Address: 18240080
Start tick: 540625
End tick: 540625
Heap size: 3980
```

The address is given as a hexadecimal number; Requested, Really Allocated and Heap size are given in bytes; Start tick and End tick are (currently) the millisecond value of the system time.

When "memory size" output is specified, the program prints a set of lines on `stdout` that look like:

```
84      84      70      3980
124     124     98      3980
```

The first column gives the total memory allocated (to give a 'time in megabytes'); the second column gives the total live memory used; the third column gives the total live memory that would be used given no internal fragmentation (i.e. only the requested size was allocated, and no more); the fourth column gives the heap size.

This output can be redirected into a file (using a command such as `./processtracefile m` *tracefile* `>` *file*) which can be fed into graphing software (section B.2).

When "object size" output is specified, the program prints a set of lines on `stdout` that look like:

```
0       46
4       1028
5       1949
```

The first column gives the object size in bytes, and the second column gives the frequency of allocation. This can also be redirected into a file that can then be fed into some graphing software.

B.2 Producing graphs

The graphs were produced using `gnuplot`[1], a free graphing program available on most Unix systems. Entering `gnuplot` at a Unix shell prompt should launch the program.

To plot the object size and frequency, the command that needs to be entered is `plot '`*file*`' using 1:2`, where *file* is the name of the file containing the output of object sizes. It may be necessary to use logarithmic instead of linear scales to see the plots properly; this can be done with the command `set logscale xy`.

Plotting the memory sizes is similar, but due to having multiple output columns, a few extra commands are needed:

```
plot 'file' using 1:4 with lines title ``Heap size''
replot 'file' using 1:2 with lines title ``Live memory''
replot 'file' using 1:3 with lines title ``Total requested memory''
```

The `replot` command tells `gnuplot` to add plots to an existing graph; the `with lines` qualifier joins the point with lines, instead of just plotting individual points.

B.3 Endian issues

If the processing of the trace files is attempted on a machine that does not use an x86-type processor, such as a Sun SPARC or Apple Mac (which uses a PowerPC processor), then the result of processing the trace files will look like garbage. This is due to the way the different architectures store data (outlined in section 4.5.6), and so byte orders need to be swapped for the traces to be usable. Included in the `MyHeap` code is a file called `switchendian.c` which does this. Compilation (at a Unix shell prompt) is achieved with the command `cc -o switchendian switchendian.c`.

Running the program requires specifying a trace file on the command line (`./switchendian` *tracefile*). A new file is generated, *tracefile*`.end`, which contains the byte-swapped traces. This file can then be fed into the tracefile processor successfully.

B.4 Multiple heaps issues

Due to the applications in the Symbian OS creating multiple heaps during the course of their execution, and the trace file processor not being able to keep

[1] `http://www.gnuplot.info`

track of multiple heaps, another program is included with the `MyHeap` code, called `getheapops.c`. This runs over a tracefile, picking up any operations that are performed on the first heap that is created in an application run, and writing them out to a new tracefile. Compilation (at a Unix shell prompt) is achieved with the command `cc -o getheapops getheapops.c`.

Running the program requires specifying a trace file on the command line (as with the `switchendian` program), and a new file is created with the name of the file that was given on the command line, with `.oneheap` appended to the filename.

Bibliography

[1] A. Bohra and E. Gabber. Are mallocs free of fragmentation? 2001.

[2] R. Harrison. *Symbian OS C++ for Mobile Phones*, chapter Foreword. Wiley, 2003.

[3] R. Harrison. *Symbian OS C++ for Mobile Phones*, chapter 3.2. Wiley, 2003.

[4] R. Harrison. *Symbian OS C++ for Mobile Phones*, chapter 1.2. Wiley, 2003.

[5] R. Harrison. *Symbian OS C++ for Mobile Phones*, chapter 6.2.6. Wiley, 2003.

[6] R. Harrison. *Symbian OS C++ for Mobile Phones*, chapter 2.4. Wiley, 2003.

[7] R. Harrison. *Symbian OS C++ for Mobile Phones*, chapter 1.1. Wiley, 2003.

[8] M. S. Johnstone and P. R. Wilson. The memory fragmentation problem: Solved? In *Proceedings of the Intl. Symp. on Memory Management (ISMM)*, pages 26–36, October 1998.

[9] P.-H. Kamp. Malloc(3) revisited. In *Usenix 1998 Annual Technical Conference: Invited Talks and Freenix Track*, pages 193–198. Usenix, June 1998.

[10] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*, chapter 5.4. Prentice-Hall, first edition, 1978.

[11] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*, chapter 8.7. Prentice-Hall, first edition, 1978.

[12] K. C. Knowlton. A fast storage allocator. *Communications of the ACM*, 8(10):623–625, 1965.

[13] D. Knuth. *The Art of Computer Programming*, volume 1: Fundamental Algorithms, chapter unknown. Addison-Wesley, 1973.

[14] D. Knuth. *The Art of Computer Programming*, volume 1: Fundamental Algorithms, chapter unknown. Addison-Wesley, 1973.

[15] J. L. Peterson and T. A. Norman. Buddy systems. *Communications of the ACM*, 20(6):421–431, 1977.

Results

On the following pages are all the graphs that have been obtained from processing the trace files.

Code

The following pages contain a listing of all the code that was used in the project. Please note that some of the code and comments in the `MyHeap` code have been taken from the Symbian allocator implementation – in these files I have indicated my own code by using a mark in the margin. The C code (for processing the trace files) is entirely my own work.