Memory Management Overview

- **Purpose**
  Provides functionality concerned with how memory is allocated for and within programs. Fundamental to Symbian OS programs is a concern that memory, as a limited resource, is carefully handled, particularly in the event of error conditions. For this reason, exception handling and memory management are closely tied together in the Cleanup Support API.

- **Architectural relationships**
  The Uikon framework uses these APIs to provide to each GUI program the basic infrastructure for well-behaved memory handling. In particular, each GUI program has support for clearing up memory in exception conditions, and, in debug builds, detection of memory leaks.

Memory Management Concepts

- **Address Space and Process**
- **Chunks**
- **Heaps**
- **Structure of a heap**
- **Virtual machine model**
- **Types of Memory in Symbian OS**

Address Space and Process

- **Programs in Symbian OS consist of a number of processes**, each of which contains one or more conceptually concurrent threads of execution.
- **Each user process has its own private address space**, i.e. a collection of memory regions which that process can access. A user process cannot directly address memory areas in the address space of another process. Threads belonging to a user process run at user privilege level.
- **There is a special process**, the Kernel process, whose threads run at supervisor privilege level. This process normally contains two threads:
  - the Kernel server thread, which is the initial thread whose execution begins at the reset vector, and which is used to implement all kernel functions requiring allocation or deallocation on the Kernel heap
  - the null thread, which runs only when no other threads are ready to run. The null thread places the processor into idle mode to save power.
Address Space and Process

- The address space of a process consists of a number of chunks, where a chunk is a region of RAM mapped into contiguous virtual addresses. On creation, a user process contains one thread (the main thread) and one to three chunks; these are:
  - The stack/heap chunk containing the stack and the heap used by the main thread of the process; this chunk always exists.
  - The code chunk; this exists only if the process is loaded into RAM (i.e. it is not ROM resident).
  - The data chunk; this exists only if the process has static data.
- If a process creates additional threads, then a new chunk is created for each new thread. Each chunk contains the thread's stack; if a new thread is not sharing an existing heap, then the chunk also contains a new heap.

Chunks

- Chunks map RAM or memory-mapped I/O devices into contiguous virtual addresses.
- A chunk consists of a reserved region and a committed region. The reserved region is the contiguous set of virtual addresses accessible to running code. The committed region is the region in which RAM (or memory-mapped I/O) is actually mapped. The size of a chunk is dynamically alterable, allowing the committed region to vary in size from zero up to the reserved region size, in integer multiples of the processor page size. This allows processes to obtain more memory on demand. Generally the committed region starts at the bottom of the reserved region.

Chunk

- A chunk also has a maximum size, which is defined when the chunk is created. The reserved region can be smaller than this maximum size, but it can also be made bigger by reallocating it. The reserved region cannot be made bigger than the maximum size.
- The size of the reserved region of a chunk is always an integer multiple of the virtual address range of a single entry in the processor page directory (PDE size). This means that the reserved region of a chunk is mapped by a number of consecutive page directory entries (PDEs). Any given PDE maps part of the reserved region of at most one chunk.
- Symbian OS has a number of chunk types, but for user side code, the chunks of interest are User chunks and Shared chunks.

User chunks

- **User chunks**
- On systems with an MMU, Symbian OS provides three types of user chunks. Each type is characterised by having a different subset of the reserved address range containing committed memory:
  - Normal chunks
  - Double-ended chunks
  - Disconnected chunks

Normal chunks

- These chunks have a committed region consisting of a single contiguous range starting at the chunk’s base address and a size that is a multiple of the MMU page size.
Normal chunks diagram

Double-ended chunks

These chunks have a committed region consisting of a single contiguous range starting at arbitrary lower and upper endpoints within the reserved region. The only condition is that the lower and upper endpoints must be a multiple of the MMU page size. Both the bottom and top of the committed region can be altered dynamically.

Disconnected chunks diagram

Disconnected chunks

These chunks have a committed region consisting of an arbitrary set of MMU pages within the reserved region. Each page-sized address range within the reserved region starting on a page boundary can be committed independently.

Disconnected chunks diagram

Shared chunks

• A shared chunk is a mechanism that allows kernel-side code to share memory buffers safely with user-side code. By kernel-side code, we usually mean device driver code.
• The main points to note about shared chunks are:
  – They can only be created and destroyed by device drivers. It is typical behaviour for user-side code, which in this context we refer to as the client of the device driver, to pass a request to the device driver to open a handle to a shared chunk. This causes the device driver to open a handle to the chunk and return the handle value to the client. Successful handle creation also causes the chunk’s memory to be mapped into the address space of the process to which the client’s thread belongs. Note, however, that it is the device driver that dictates exactly when the chunk itself is created, and when memory is committed. The precise protocol depends on the design of the driver; you need to refer to that driver’s documentation for programming guidance.
Shared chunks

- Like all kernel-side objects, a shared chunk is reference counted. This means that it remains in existence as long as the reference count is greater than zero. Once all opened references to the shared chunk have been closed, regardless of whether the references are user-side, or kernel-side, then it is destroyed.
- User-side code that has gained access to a shared chunk from one device driver can pass this to a second device driver. The second device driver must open the chunk before it can be used.
- More than one user-side application can access the data in a shared chunk. A handle to a shared chunk can be passed from one process to another process using standard handle passing mechanisms. In practice, handles would be passed in a client-server context, either from client to server or from server to client using inter-process communication (IPC).
- Processes that share data inside a chunk should communicate the location of that data as an offset from the start of the chunk, and not as an absolute address. The shared chunk may appear at different addresses within the address spaces of different user processes.

Local and global chunks

- **Local chunks**
  - A chunk is local when it is private to the process creating it and is not intended for access by other user processes.
  - A local chunk cannot be mapped into any other process and is, therefore, used for memory that does not need to be shared.
  - A local chunk does not have a name.

- **Global chunks**
  - A chunk is global if it is intended to be accessed by other processes.
  - Global chunks have names that can be used to identify the chunk to another process wishing to access it. A process can open a global chunk by name; this maps the chunk into that process's address space, allowing direct access and enabling the sharing of data between processes.

Heaps

- Each thread has a chunk which contains that thread's program stack. For the main thread of a process, this chunk also contains the thread's heap. A program's request for memory is allocated from this heap.
- If a process creates additional threads, then a new chunk is created for each new thread. Each chunk contains the thread's stack; if a new thread is not sharing an existing heap, then the chunk also contains a new heap.
- When a new thread is created, either:
  - a new heap is created for it
  - it uses the creating thread's heap
  - it uses an explicitly referenced heap.

Structure of a heap

- A heap simply consists of two lists of cells; one is the list of allocated cells and the other the list of free cells. Each list is anchored in the heap object.
- A cell consists of a cell header followed by the body of the cell itself. The body of the cell is the area of memory which is considered allocated.

Using Memory Allocation

- How to share heaps
- How to switch heaps
- How to walk the heap
How to share heaps

- Heaps may be shared between threads within a process.
- When a new thread is created:
  - it can use the same heap as the thread which is doing the creating.
  - it can use the heap which has been explicitly created for it by the thread which is doing the creating.
  - it can use the heap automatically created for it by the operating system.
  - Only in the first two cases is the heap being shared.

How to switch heaps

- When freeing memory, it is extremely important that the current heap is the heap from which it was allocated. Trying to free a memory cell from a heap when it was allocated from a different heap has undefined consequences.
  A thread can switch between heaps using User::SwitchHeap(). After a call to this function, any new request for memory is satisfied from the new heap.

How to walk the heap

- A heap can be checked to make sure that its allocated and free cells are in a consistent state and that the heap is not corrupt.

Virtual machine model

The Kernel provides a virtual machine environment to user processes. Each process accesses its data in the same virtual address range, called the data section, which ranges from:

0x00400000 to 0x3FFFFFFF

but note that the static data always appears at:

0x00400000

The code chunk for RAM loaded processes always appears at:

0x20000000

This allows multiple processes to run, each executing the same code (e.g., multiple word documents open at the same time, each in a separate instance of the word application) where the same code chunk is used for each of the processes. This reduces RAM usage.

In effect, each user process has the same kind of view. Code instructions address data using the virtual address; the Memory Management Unit (MMU) is responsible for the translation of the virtual address to the physical RAM address.

Only one chunk can occupy a given virtual address range at a time, so a context switch between different processes involves re-mapping the chunks. The process chunks of the old process are re-mapped to their home addresses. These are in the home section, which is the virtual address range from:

0x80000000 to 0xFFFFFFFF

ROM code is normally mapped into the address range:

0x50000000 to 0x5FFFFFFF
Virtual machine model

- The process chunks of the new process are mapped from their home addresses back to the data section. Chunks which are not accessible by the current user process reside in the home section, and they have supervisor mode only access permissions, so that only the kernel can access them.
- The kernel's data and stack/heap chunks also reside in the home section. These are never visible to user processes. Code chunks for RAM-loaded libraries reside at the top end of the home section and have user read-only access, so that all user processes can execute code from a loaded library.
- A context switch between processes thus involves:
  - Moving the old process chunks to the home section, and changing their access permissions to supervisor-only
  - Moving the new process chunks to the data section and changing their access permissions back to user-accessible.
- This is best seen graphically. In the first diagram shown below, user-process 1 is running and can 'see' all chunks in the clear. Dark backgrounds represent chunks which are not visible to user-process 1.

When user-process 2 is running, the context switch re-maps the user-process 2 data to the data section and user process 1 data is re-mapped to the home section as the second diagram shows.

Virtual Memory and Paging

- The two memory areas of interest are the home area and the run area.
Types of Memory in Symbian OS

Direct Memory Access

• Direct Memory Access (DMA) is used by Symbian OS to offload the burden of high bandwidth memory to peripheral data transfers. DMA can reduce the interrupt load by a factor of 100 for a peripheral, by moving and increasing the real-time robustness of that interface.

• A DMA engine is a peripheral device. It can be programmed as many channels as peripheral ports that require DMA, and an additional channel for memory-to-memory transfers. This channel can be used to move large quantities of data between peripherals and memory without the intervention of the CPU.

• Multi-channel DMA engines are capable of handling more than one peripheral at a time. Symbian OS has the advantage of being able to load DMA engines for one peripheral without the need for additional DMA engines.

• A DMA channel will be initiated by programming the DMA engine with a list of DMA commands. These commands may include start and stop transfers, transfers to and from memory, and transfers to and from peripheral registers. DMA engines can be configured to perform a variety of tasks, including data transfer, memory management, and peripheral control.

Memory Areas

• The home area is where the chunks of memory for a process are kept when the process has been launched but is not currently active processes. The home area is a protected region of memory – only kernel-level code can read and write it. When a process is scheduled to execute, its data chunks are moved (immigrated) from the home area to the peripheral FIFO. The channel will increment the read address pointer by the size of the channel to bring the next DMA command into the engine. At the end of each transfer, the DMA engine will load the next command – until it runs out of commands. New commands can be added to the end of the list while DMA is in progress, so in theory an audio example need never stop.

• A DMA engine that saves an interrupt at the end of every transfer is single-buffered. The CPU will have to service an interrupt at the end of the command. Double-buffering increases the read buffer capacity window up to the duration of a whole transfer, for example about 20 ms for a 4 KB audio transfer buffer.

Types of Flash Disk

• Internal Flash Disk

- The internal Flash disk is a single disk drive and allows for reading and writing files to it. It contains the Symbian OS and all user data. The Symbian OS expects a hierarchical directory structure, with very similar features to those you would find on a file-based operating system. The internal Flash disk is protected against writes when the system is in shutdown mode, and the home area as the backstage area. When a process is ready to perform - I.e., its data chunks are moved to the run area, and the process executes.

- Why aren't the process data chunks simply left in the home area when the process executes? The reason is that the process code always expects its data to reside in the same place. This is because the process data chunks are always running in a single run area, which contains references to static variables. This is also the case for code chunks that are never moved to the run area. For example, code chunks are never moved to the run area.

• Removable memory cards

- Memory cards are removably disk drives and allow you to expand the storage provided internally. You can also add them to read or write to the memory card. For example, you can read music files from a memory card and use the Symbian OS file system.

- Memory cards can be represented as additional disk volumes on your smartphone’s file system. They can be formatted (MMC and SD are examples) and available sizes vary by phone. Memory card sizes can vary from 16MB for even less to 128GB.

Random Access Memory (RAM)

• RAM is volatile execution and data memory used by running programs. Applications vary in how much RAM they use, and this depends on what the application is doing at the time. For example, a browser application loading a web page needs to allocate more RAM than RAM to load user data. Also, the more programs you run, the more programs you can run on your smartphone at once. Typically, mobile phones have between 7 and 30MB of RAM available for applications to use.

Types of Random Access Memory

• Read Only Memory (ROM)

- The ROM is where the Symbian OS software itself resides. It includes all the startup code to boot the device, as well as all device drivers and other hardware-specific code. This area cannot be written to by a user, although some can be overwritten by the applications that use it. Depending on the application, the ROM may consist of the following:

- The Symbian OS software itself resides in the ROM.

- Various peripheral drivers, such as those for the audio, keyboard, and display.

- The file system, which allows you to save user data, and even install applications. This card is treated as storage provided internally. You can also read from and write to a memory card like a floppy disk.

- Program metadata and the reading of the ECC registers for each incoming channel. They are also good for complex peripheral interactions where a DMA engine can add more instructions to the peripheral interface.
Flash memory

- Symbian phones use Flash memory as their principal store of system code and user data. Flash memory is a silicon-based non-volatile storage medium that can be programmed and erased electronically. Flash memory comes in two major types: NOR and NAND. The names refer to their fundamental silicon gate design. Symbian OS phones make best use of both types of Flash through the selection of file systems. The built-in system code and applications appear to Symbian software as one large read-only drive, known as the Z: drive. The Z: drive is sometimes known as the ROM image. User data and installed applications reside on the internal, writable C: drive. A typical Symbian phone today will use between 32 and 64 MB of Flash for the code and user data – this is the total ROM budget. Symbian uses many techniques to minimize the code and data sizes within a phone, such as THUMB instruction set, prelinked XIP images, compressed executables, compressed data formats and coding standards that emphasize minimal code size.

Flash memory

- NOR Flash
  - NOR flashes allow for unlimited writes to the same data block, to turn the ones into zeros. Flashes usually have a write buffer of around 32 to 64 bytes that allows a number of bytes to be written in parallel to increase speed. Erasing a NOR segment is slow, taking about half a second to one second. But erases can be suspended and later restarted. Completed writes and erases will update the status register within the Flash, and may generate an external interrupt. Without an interrupt, the CPU will need to use a high-speed timer to poll the Flash for completion.

- NAND Flash
  - NAND Flash is treated as a block-based disk, rather than randomly addressable memory. Unlike NOR, it does not have any address lines, so cannot appear in the memory map. This means that code cannot execute directly from NAND and it has to be copied into RAM first. This results in the need for extra RAM in a NAND phone compared to a similar NOR device. NAND Flash writes are about 10 times faster than those on NOR Flash. A phone cannot boot directly from NAND. The process is more complex, requiring a set of boot loaders that build upon each other, finally resulting in a few megabytes of core Symbian OS image, the ROM, being loaded into RAM, where it will execute. NAND is also inherently less reliable than NOR. The lower price of NAND compared to NOR makes it attractive for mass-market phone projects, even after taking into account the extra RAM required.

How to track down memory leaks

- This article presents a method of tracking down a memory leak in cases where the address of that heap allocation keeps changing. The solution consists of the following steps:
  - Start the application and run it until it panics.
  - Take note of the address that wasn’t deallocated. (Assume it’s 0x1046a432.)
  - Restart the application and run it until it panics. Make sure you run it exactly as you did the first time.
  - Again take note of the address. (Assume it’s 0x1a52a432.)
  - Run it one more time, again until it panics and in exactly the same way.
  - Again take note of the address. (Assume it’s 0x245ea432.)
  - Now looking at the numbers you will find a correlation.

Random Access Memory

- Random Access Memory (RAM) is the place of all the live data within the system, and often the executing code. The quantity of RAM determines the type and number of applications you can run simultaneously, the access speed of the RAM contributes to their performance. Multimedia uses lots of RAM for megapixel cameras images and video recording. The RAM chip is a significant part of the total cost of a phone.

Random Access Memory

- Mobile SDRAM
  - Manufacturers have started to produce RAM specifically for the mobile phone market, known as Low Power or Mobile SDRAM. This memory has been optimized for lower power consumption and slower interface speeds of about 100 MHz, compared to normal PC memory that is four times faster. Mobile memories have additional features to help maintain battery life. Power down mode enables the memory controller to disable the RAM part without the need for external control circuitry.

- Internal RAM (IRAM)
  - Memory that is embedded within the System-on-Chip (SoC) is known as Internal RAM (IRAM). This is much less than that of main memory. IRAM can be used as an internal frame buffer. An LCD controller driving a dumb display needs to re-read the entire frame buffer 60 times a second. IRAM can also be useful as a scratch pad or message box between multiple processors on the SoC.