(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
3 December 2009 (03.12.2009)

PCT

(10) International Publication Number
WO 2009/144384 A1

(51) International Patent Classification:
G06F 12/02 (2006.01) *G06F 12/12* (2006.01)
G06F 12/08 (2006.01)

(21) International Application Number:
PCT/FI2009/050461

(22) International Filing Date:
1 June 2009 (01.06.2009)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
0809926.9 30 May 2008 (30.05.2008) GB

(71) Applicant *(for all designated States except US)*: **NOKIA CORPORATION** [FI/FI]; Keilalahdentie 4, FI-02150 Espoo (FI).

(72) Inventor; and

(75) Inventor/Applicant *(for US only)*: **MEDHURST, Jonathan** [GB/GB]; 17 Noel Street, Leicester, LE03 0df, LONDON (GB).

(74) Agent: **NOKIA CORPORATION**; IPR Department, Virpi Tognetty, Keilalahdentie 4, FI-02150 Espoo (FI).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IS, JP, KE, KG, KM, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LT, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PG, PH, PL, PT, RO, RS, RU, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:

— *with international search report (Art. 21(3))*

— *before the expiration of the time limit for amending the claims and to be republished in the event of receipt of amendments (Rule 48.2(h))*

(54) Title: MEMORY PAGING CONTROL METHOD AND APPARATUS

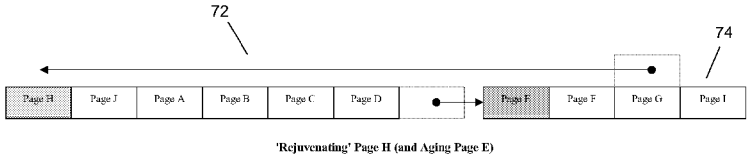


Figure 7

(57) Abstract: A page replacement algorithm which provides substantially the benefits of an LRU algorithm but with the implementational simplicity of a FIFO arrangement is described. In particular, pages which have been paged into memory are maintained in a paging cache, which is split into at least two parts. Each part is maintained as a separate FIFO, and new pages are loaded into the first part, with pages output from the first part being placed into the second part. Hence, essentially the arrangement is no more complex than simply maintaining two FIFO lists. However, to ensure that pages are not immediately deleted from the cache when they reach the output of the first part of the cache, they are retained in the second part of the cache, and whilst they are working their way through the second part of the cache they can be re-instated into the input of the first part of the cache whenever required.

WO 2009/144384 A1

Memory Paging Control Method and Apparatus

Technical Field

Embodiments of the present invention relate to a memory paging control method and apparatus. In some embodiments the invention relates to such a method and apparatus for controlling which pages are maintained in a paging cache in the executable memory of a computing device.

Background to the Invention

Known techniques for controlling which pages are maintained in the cache of a computing device suffer from certain disadvantages. It is an aim of the present invention to provide an improved technique for loading and caching pages of memory.

Summary of the Invention

In a first example of the invention there is provided a method comprising: maintaining a paging cache of memory pages which have been paged into random access memory (RAM) for use by a processor, the paging cache being arranged in at least a first part and a second part, each part being configured to operate on a respective First-In, First-Out (FIFO) basis; loading a new page into RAM, the new page being loaded into the first part of the cache; outputting a page from the first part of the cache into the second part of the cache; and when a page is required which is located in the second part of the cache, removing the required page from the second part of the cache and placing the required page into the first part of the cache.

In a further embodiment there is provided apparatus comprising: a processor; and a paging cache of memory pages which have been paged into random access memory (RAM), the paging cache being arranged in at least a first part and a second part, each part being configured to operate on a respective First-In, First-Out (FIFO) basis; wherein the processor is configured to cause the apparatus to perform the following:- i) load a new page into RAM, the new page being loaded into the first part of the cache; ii) output a page from the first part of the cache into the second part of the cache; and iii)

when a page is required which is located in the second part of the cache, remove the required page from the second part of the cache and place the required page into the first part of the cache.

In further embodiments the invention provides apparatus comprising: processor means; and paging cache means for storing memory pages which have been paged into random access memory (RAM), the paging cache means being arranged in at least a first part and a second part, each part being configured to operate on a respective First-In, First-Out (FIFO) basis; wherein the processor means is configured to cause the apparatus to perform the following:- i) load a new page into RAM, the new page being loaded into the first part of the cache; ii) output a page from the first part of the cache into the second part of the cache; and iii) when a page is required which is located in the second part of the cache, remove the required page from the second part of the cache and place the required page into the first part of the cache.

The processor means may include one or more separate processor cores. The paging cache means may be provided in any suitable type of memory.

In other examples, the invention may include a computer program, a suite of computer programs, a computer readable storage medium, or any software arrangement for implementing the method of the first example. Aspects of the invention may also be carried out in hardware, or in a combination of software and hardware.

Brief Description of the Drawings

Features and advantages of example embodiments of the present invention will become apparent from the following description with reference to the accompanying drawings, wherein: -

Figure 1 is a block diagram of a smartphone architecture;

Figure 2A is a diagram illustrating a memory layout forming background to the invention;

Figure 2B is a diagram illustrating a memory layout forming background to the invention;

Figure 2C is a diagram illustrating a memory layout according to an embodiment of the invention;

Figure 3 is a diagram illustrating how paged data can be paged into RAM;

Figure 4 is a diagram illustrating a paging cache;

Figure 5 is a diagram illustrating how a new page can be added to the paging cache;

Figure 6 is a diagram illustrating how pages can be aged within a paging cache;

Figure 7 is a diagram illustrating how aged pages can be rejuvenated in a paging cache;

Figure 8 is a diagram illustrating how a page can be paged out of the paging cache;

Figure 9 is a diagram illustrating the RAM savings obtained using demand paging;

Description of the Embodiments

Figure 1 shows an example of a device that may benefit from embodiments of the present invention. The smartphone 10 comprises hardware to perform the telephony functions, together with an application processor and corresponding support hardware to enable the phone to have other functions which are desired by a smartphone, such as messaging, calendar, word processing functions and the like. In Figure 1 the telephony hardware is represented by the RF processor 102 which provides an RF signal to antenna 126 for the transmission of telephony signals, and the receipt therefrom. Additionally provided is baseband processor 104, which provides signals to and receives signals from the RF Processor 102. The baseband processor 104 also interacts with a subscriber identity module 106.

Also provided are a display 116, and a keypad 118. These are controlled by an application processor 108, which is often a separate integrated circuit from the baseband processor 104 and RF processor 102. A power and audio controller 120 is provided to supply power from a battery to the telephony subsystem, the application processor, and the other hardware. Additionally, the power and audio controller 120 also controls input from a microphone 122, and audio output via a speaker 124.

In order for the application processor 108 to operate, various different types of memory are often provided. Firstly, the application processor 108 is provided with some Random Access Memory (RAM) 112 into which data and program code can be written

and read from at will. Code placed anywhere in RAM can be executed by the application processor 108 from the RAM.

Additionally provided is separate user memory 110, which is used to store user data, such as user application programs (typically higher layer application programs which determine the functionality of the device), as well as user data files, and the like.

Many modern electronic devices make use of operating systems. Modern operating systems can be found on anything composed of integrated circuits, like personal computers, Internet servers, cell phones, music players, routers, switches, wireless access points, network storage, game consoles, digital cameras, DVD players, sewing machines, and telescopes. An operating system is the software that manages the sharing of the resources of the device, and provides programmers with an interface to access those resources. An operating system processes system data and user input, and responds by allocating and managing tasks and internal system resources as a service to users and programs on the system. At its most basic, the operating system performs tasks such as controlling and allocating memory, prioritising system requests, controlling input and output devices, facilitating networking, and managing files. An operating system is in essence an interface by which higher level applications can access the hardware of the device.

In order for the application processor 108 to operate in the embodiment of Figure 1, an operating system is provided, which is started when the smartphone system 10 is first switched on. The operating system code is commonly stored in a Read-Only Memory, and in modern devices, the Read-Only Memory is often NAND Flash ROM 114. The ROM will store the necessary operating system component in order for the device 10 to operate, but other software programs may also be stored, such as application programs, and the like, and in particular those application programs which are mandatory to the device, such as, in the case of a smartphone, communications applications and the like. These would typically be the applications which are bundled with the smartphone by the device manufacturer when the phone is first sold. Further applications which are added to the smartphone by the user would usually be stored in the user memory 110.

ROM (Read-Only Memory) traditionally refers to memory devices that physically store data in a way which cannot be modified. These devices also allow direct random access to their contents and so code can be executed from them directly - code is eXecute-In-Place (XIP). This has the advantage that programs and data in ROM are always available and don't require any action to load them into memory. The term ROM can be used to mean 'data stored in such a way that it behaves like it is stored in read-only memory'. The underlying media may actually be physically writeable, like RAM or flash memory but the file system presents a ROM-like interface to the rest of the OS, for example as a particular drive.

The ROM situation is further complicated when the underlying media is not XIP. This is the case for NAND flash, used in many modern devices. Here code in NAND is copied (or shadowed) to RAM, where it can be executed in place. One way of achieving this is to copy the entire ROM contents into RAM during system boot and use the Memory Management Unit (MMU) to mark this area of RAM with read-only permissions. The data stored by this method is called the Core ROM image (or just Core image) to distinguish it from other data stored in NAND. The Core image is an XIP ROM and is usually the only one; it is permanently resident in RAM.

Figure 2, layout A shows how the NAND flash 20 is structured in a simple example. All the ROM contents 22 are permanently resident in RAM and any executables in the user data area 24 (for example the C: or D: drive) are copied into RAM as they are needed.

The above method can be costly in terms of RAM usage, and a more efficient scheme can be used to split the ROM contents into those parts required to boot the OS, and everything else. The former is placed in the Core image as before and the latter is placed into another area called the Read-Only File System (ROFS). Code in ROFS is copied into RAM as it is needed at runtime, at the granularity of an executable (or other whole file), in the same way as executables in the user data area. In a specific example of an embodiment using Symbian OS, the component responsible for doing this is the 'Loader', which is part of the File Server process.

In an example embodiment, there are several ROFS images, for example localisation and/or operator-specific images. Usually, the first one (called the primary ROFS) is

combined with the Core image into a single ROM-like interface by what is known as the Composite File System.

Layout B in Figure 2 shows a Composite File System structure of another example. Here, ROM 30 is divided into the Core Image 32 comprising those components of the OS which will always be loaded into RAM, and the ROFS 34 containing those components which do not need to be continuously present in RAM, but which can be loaded in and out of RAM as required. As mentioned, components in the ROFS 34 are loaded in and out of RAM as whole components when they are required (in the case of loading in) or not required. Comparing this to layout A, it can be seen that layout B is more RAM-efficient because some of the contents of the ROFS 34 are not copied into RAM at any given time. The more unused files there are in the ROFS 34, the greater the RAM saving.

It would, however, be beneficial if even further RAM savings could be made. Virtual memory techniques are known in the art, where the combined size of any programs, data and stack exceeds the physical memory available, but programs and data are split up into units called pages. The pages which are required to be executed can be loaded into RAM, with the rest of the pages of the program and data stored in non XIP memory (such as on disk). Demand paging refers to a form of paging where pages are loaded into memory on demand as they are needed, rather than in advance. Demand paging therefore generally relies on page faults occurring to trigger the loading of a page into RAM for execution.

An example embodiment of the invention to be described is based upon the smartphone architecture shown in Figure 1, and in particular a smartphone running Symbian OS. Within Symbian OS, as mentioned, the part of the operating system which is responsible overall for loading programs and data from non XIP memory into RAM is the "loader". Many further details of the operation of the loader can be found in Sales J. *Symbian OS Internals* John Wiley & Sons, 2005, and in particular chapter 10 thereof., the entire contents of which are incorporated herein be reference. Within the example embodiment to be described the operation of the loader is modified to allow demand paging techniques to be used within the framework of Symbian OS.

In particular, according to the example embodiment, a smartphone is provided having a composite file system as previously described, wherein the CFS provides a Core Image comprising those components of the OS which will always be loaded into RAM, and the ROFS containing those components which do not need to be continuously present in RAM, but which can be loaded in and out of RAM as required. In order to reduce the RAM requirement of the smartphone, within the example embodiment the principles of virtual memory are used on the core image, to allow data and programs to be paged in and out of memory when required or not required. By using virtual memory techniques such as this, then RAM savings can be made, and overall hardware cost of a smartphone reduced.

Since an XIP ROM image on NAND is actually stored in RAM, an opportunity arises to demand page the contents of the XIP ROM, that is, read its data contents from NAND flash into RAM (where it can be executed), on demand. This is called XIP ROM Paging (or demand paging)."Paging" can refer to reading in required segments ("pages") of executable code into RAM as they are required, at a finer granularity than that of the entire executable. Typically, page size may be around 4kB; that is, code can be read in and out of RAM as required in 4kB chunks. A single executable may comprise a large number of pages. Paging is therefore very different from the operation of the ROFS, for example, wherein whole executables are read in and out of RAM as they are required to be run.

In the example embodiment of the invention an XIP ROM image is split into two parts, one containing unpaged data and one containing data paged on demand. In this example the unpaged data is those executables and other data which cannot be split up into pages. The unpaged data consists of kernel-side code plus those parts that should not be paged for other reasons (e.g. performance, robustness, power management, etc). The terms 'locked down' or 'wired' can also be used to mean unpaged. Paged data in this example is those executables and other data which can be split up into pages.

At boot time, the unpaged area at the start of the XIP ROM image is loaded into RAM as normal but the linear address region normally occupied by the paged area is left unmapped - i.e. no RAM is allocated for it in this example.

When a thread accesses memory in the paged area, it takes a page fault. The page fault handler code in the kernel then allocates a page of RAM and reads the contents for this from the XIP ROM image contained on storage media (e.g. NAND flash). As mentioned, a page is a convenient unit of memory allocation: in this example it is 4kB. The thread then continues execution from the point where it took the page fault. This process is referred to in this example embodiment as 'paging in' and is described in more detail later.

When the free RAM on the system reaches zero, memory allocation requests can be satisfied by taking RAM from the paged-in XIP ROM region. As RAM pages in the XIP ROM region are unloaded, they are 'paged out'. Figure 3 shows the operations just described.

Note that the content in the paged data area of an XIP ROM is subject to paging in this example, not just executable code; accessing any file in this area may induce a page fault. A page may contain data from one or more files and page boundaries do not necessarily coincide with file boundaries in the example embodiment.

Figure 2, layout C shows an XIP ROM paging structure according to the example embodiment. Here, ROM 40 comprises an unpaged core area 42 containing those components which should not be paged, and a paged core area 44 containing those components which should reside in the core image rather than the ROFS, but which can be paged. ROFS 46 then contains those components which do not need to be in the Core image. Although the unpaged area of the Core image may be larger than the total Core image in layout B, only a fraction of the contents of the paged area needs to be copied into RAM compared to the amount of loaded ROFS code in layout B.

Further details of the algorithm which controls demand paging in this example embodiment will now be described. All memory content that can be demand paged is said in this example to be 'paged memory' and the process is controlled by the 'paging subsystem'. Other terms that are used in describing example embodiments of the invention are:

1. Live Page - A page of paged memory whose contents are currently available.

2. Dead Page - A page of paged memory whose contents are not currently available.
3. Page In - The act of making a dead page into a live page.
4. Page Out - The act of making a live page into a dead page. The RAM used to store the content of this may then be reused for other purposes.

In one embodiment, efficient performance of the paging subsystem is dependent on the algorithm that selects which pages are live at any given time, or conversely, which live pages should be made dead. The paging subsystem of this embodiment approximates a Least Recently Used (LRU) algorithm for determining which pages to page out. The memory management unit 28 (MMU) provided in the example device is a component comprising hardware and software which has overall responsibility for the proper operation of the device memory, and in particular for allowing the application processor to write to or read from the memory. The MMU is part of the paging subsystem of this example embodiment.

The paging algorithm according to the present embodiment provides a "live page list". All live pages are stored on the 'live page list', which is a part of the paging cache. Figure 4 shows the live page list. The live page list is split into two sub-lists, one containing young pages (the "young page list" 72) and the other, old pages (the "old page list" 74). The memory management unit (MMU) 58 in the device of this example is used to make all young pages accessible to programs but the old pages inaccessible. However, the contents of old pages are preserved and they still count as being live. The net effect is of a FIFO (first-in, first-out) list in front of an LRU list, which results in less page churn than a plain LRU.

Figure 5 shows what happens when a page is "paged in" in this example embodiment. When a page is paged in, it is added to the start of the young list 72 in the live page list, making it the youngest.

The paging subsystem of some embodiments attempts to keep the relative sizes of the two lists equal to a value called the young/old ratio. If this ratio is `R`, the number of young pages is `Ny` and the number of old pages is `No` then if (`Ny > RNo`), a page is

taken from the end of the young list 72 and placed at the start of the old list 74. This process is called ageing, and is shown in Figure 6.

If an old page is accessed by a program in an example embodiment, this causes a page fault because the MMU has marked old pages as inaccessible. The paging subsystem then turns that page into a young page (i.e. rejuvenates it), and at the same time turns the last young page into an old page. This is shown in Figure 7, wherein the old page to be accessed is taken from the old list 74 and added to the young list 72, and the last (oldest) young page is aged from the young list 72 to the old list 74.

When the operating system requires more RAM for another purpose then it may obtain the memory used by a live page. In one example the 'oldest' live page is selected for paging out, turning it into a dead page, as shown in Figure 8. If paging out leaves too many young pages, according to the young/old ratio, then the last young page (e.g. Page D in Figure 8) would be aged. In this way, the young/old ratio helps to maintain the stability of the paging algorithm, and ensure that there are always some pages in the old list.

When a program attempts to access paged memory that is 'dead', a page fault is generated by the MMU and the executing thread is diverted to the Symbian OS exception handler. This performs the following tasks:

1. Obtain a page of RAM from the system's pool of unused RAM (i.e. the 'free pool'), or if this is empty, page out the oldest live page and use that instead.
2. Read the contents for this page from some media (e.g. NAND flash).
3. Update the paging cache's live list as described previously.
4. Use the MMU to make this RAM page accessible at the correct linear address.
5. Resume execution of the program's instructions, starting with the one that caused the initial page fault.

In some embodiments the above actions are executed in the context of the thread that tries to access the paged memory.

When the system requires more RAM and the free pool is empty then RAM that is being used to store paged memory is freed up for use. This is referred to as 'paging out' and happens by the following process:

1. Remove the 'oldest' RAM page from the paging cache.
2. Use the MMU to mark the page as inaccessible.
3. Return the RAM page to the free pool.

Possible benefits of the demand paging algorithm of some embodiments of the invention will now be discussed. In general, a purpose of demand paging is to save RAM, but there may also be at least two other potential benefits. These benefits can be dependent on a paging configuration, discussed later.

One possible performance benefit resulting from some embodiments of the invention is due to so-called "lazy loading". In general, the cost of servicing a page fault means that paging has a negative impact on performance. However, in some cases demand paging (DP) actually improves performance compared with the non-DP composite file system case (Figure 2, layout B), especially when the use-case normally involves loading a large amount of code into RAM (e.g. when booting or starting up large applications). In these cases, the performance overhead of paging can be outweighed by the performance gain of loading less code into RAM. This is sometimes known as 'lazy loading' of code.

Note that when the non-DP case consists of a large core image (i.e. something closer to Figure 2, layout A), most or all of the code involved in a use-case may already be permanently loaded, and so the performance improvement of lazy loading may be reduced. An exception to this is during boot, where the cost of loading the whole core image into RAM contributes to the overall boot time.

A second possible performance improvement lies in improved stability of the device. The stability of a device is often at its weakest in Out Of Memory (OOM) situations. Poorly written code may not cope well with exceptions caused by failed memory allocations. As a minimum, an OOM situation will degrade the user experience.

If DP is enabled on a device and the same physical RAM is available compared with the non-DP case, the increased RAM saving makes it more difficult for the device to go OOM, avoiding many potential stability issues. Furthermore, the RAM saving achieved by DP is proportional to the amount of code loaded in the non-DP case at a particular time. For instance, the RAM saving when 5 applications are running is greater than the saving immediately after boot. This can make it even harder to induce an OOM situation.

Note that this increased stability may only apply when the entire device is OOM. Individual threads may have OOM problems due to reaching their own heap limits. DP may not help in these cases.

In addition to the above described benefits of demand paging, further performance improvements may be obtained in dependence on the demand paging configuration. In particular, demand paging can introduce three new configurable parameters to the system. These are:

1. The amount of code and data that is marked as unpaged.
2. The minimum size of the paging cache.
3. The ratio of young pages to old pages in the paging cache.

The first two are discussed below. The third should be determined empirically.

With respect to the amount of unpaged files, it is preferred in some embodiments that areas of the OS involved in servicing a paging fault are protected from blocking on the thread that took the paging fault (directly or indirectly). Otherwise, a deadlock situation may occur. This is partly achieved in Symbian OS by ensuring that all kernel-side components are always unpaged.

In addition to kernel-side components, a number of components are explicitly made unpaged in example embodiments of the invention, to meet the functional and performance requirements of a device. The performance overhead of servicing a page fault is unbounded and variable so it may be desirable to protect some critical code

paths by making files unpaged. Chains of files and their dependencies may need to be unpaged to achieve this. It may be possible to reduce the set of unpaged components by breaking unnecessary dependencies and separating critical code paths from non-critical ones.

Whilst making a component unpaged is a straightforward performance/RAM trade-off, this can be made configurable, allowing the device manufacturer in embodiments of the invention to make the decision based on their system requirements.

With respect to the paging cache size, as described previously if the system requires more free RAM and the free RAM pool is empty, then pages are removed from the paging cache in order to service the memory allocation. In some embodiments this cannot continue indefinitely or a situation may arise where the same pages are continually paged in and out of the paging cache; this is known as page thrashing. Performance is dramatically reduced in this situation.

To avoid catastrophic performance loss due to thrashing, within some embodiments a minimum paging cache size can be defined. If a system memory allocation would cause the paging cache to drop below the minimum size, then the allocation fails.

As paged data is paged in, the paging cache grows but any RAM used by the cache above the minimum size does not contribute to the amount of used RAM reported by the system. Although this RAM *is* really being used, it will be recycled whenever anything else in the system requires the RAM. So the *effective* RAM usage of the paging cache is determined by its minimum size.

In theory, it is also possible to limit the maximum paging cache size. However, this may not be useful in production devices because it prevents the paging cache from using all the otherwise unused RAM in the system. This may negatively impact performance for no effective RAM saving.

By setting a minimum paging cache size, thrashing can be prevented in some embodiments of the invention. In this respect, the minimum paging cache size relates to a minimum number of pages which should be in the paging cache at any one moment.

In one embodiment the pages in the paging cache are divided between the young list and the old list. This is not essential, however, and in other embodiments the paging cache may not be divided, or may be further sub divded into more than two lists. To help prevent thrashing, it is useful to maintain an overall minimum size of the list, and to make the pages therein accessible without having to be re-loaded into memory.

Overall the main advantage of using DP is the RAM saving which is obtained. An easy way to visualise the RAM saving achieved by DP is to compare simple configurations. Consider a non-DP ROM consisting of a Core with no ROFS (as in Figure 2, layout A). Compare that with a DP ROM consisting of an XIP ROM paged Core image, again with no ROFS (similar to Figure 2, layout C but without the ROFS). The total ROM contents are the same in both cases. Here the effective RAM saving is depicted by Figure 9.

The effective RAM saving is the size of all paged components minus the minimum size of the paging cache. Note that when a ROFS section is introduced, this calculation is much more complicated because the contents of the ROFS are likely to be different between the non-DP and DP cases.

The RAM saving can be increased by reducing the set of unpaged components and/or reducing the minimum paging cache size (i.e. making the configuration more 'stressed'). Performance can be improved (up to a point) by increasing the set of unpaged components and/or increasing the minimum paging cache size (i.e. making the configuration more 'relaxed'). However, if the configuration is made too relaxed then it is possible to end up with a net RAM increase compared with a non-DP ROM.

Demand paging is therefore able to present significant advantages in terms of RAM savings, and hence providing an attendant reduction in the manufacturing cost of a device. Additionally, as mentioned above, depending on configuration performance improvements can also be obtained.

Whilst some of the above described embodiments are discussed in the context of the example of a smartphone, it should be understood that in other embodiments different types of device may be provided, for various different functions. For example, the techniques of the present invention may be used to provide embodiments with different

applications, such as for example, as a general purpose computer, or as a portable media player, or other audio visual device, such as a camera. Any device or machine which incorporates a computing device provided with RAM into which data and programs need to be loaded for execution may benefit from the invention and constitute an embodiment thereof. The invention may therefore be applied in many fields, to provide improved devices or machines that require less RAM to operate than had heretofore been the case.

In addition, whilst embodiments have been described in respect of a smartphone running Symbian OS, which makes use of a combined file system, it should be further understood that this is presented for illustration only, and in other embodiments the concepts of the demand paging algorithms described herein may be used in other devices, and in particular devices which do not require a split file system such as the composite file system described. Instead, the demand paging algorithm herein described may be used in any device in which virtual memory techniques involving paging programs and data into memory for use by a processor may be used.

In a typical demand paging operation, when a page fault occurs and a new page is to be loaded, a decision needs to be made as to which page to remove from RAM to make room for the new page to be loaded. This is particularly the case where the RAM is full, and the number of pages in RAM cannot simply be increased.

Tanenbaum A.S. *Modern Operating Systems* 2nd ed Prentice Hall, 2001, at pages 214 to 228 describes several different page replacement algorithms which are already known in the art, such as the Not Recently Used (NRU) algorithm, Least Recently Used (LRU) algorithm, First-In First-Out (FIFO) algorithm, and the second chance algorithm.

The FIFO algorithm simply maintains a list of all pages currently in memory, with the page at the head of the list the oldest one and the page at the tail the most recent arrival. On a page fault, the page at the head of the list is removed, and the new page added to the tail of the list. Whilst a FIFO page replacement algorithm is therefore easy to implement, it is rarely used because it makes no account for whether the page that is being removed might actually be required again in the near future. In this respect, if the

removed page is required again, then a further page fault will be triggered, and the removed page will have to be re-loaded from storage into RAM.

A variation on the FIFO algorithm is the second chance algorithm. Here each page has a Read flag R, which says whether the pages has been referenced recently (i.e. in the last *n* clock cycles). The pages are maintained in a FIFO cache, but if the oldest page has its flag set such that it has been accessed recently, then when it comes to the head of the list and hence is subject to be removed, instead of being removed it is placed back at the tail of the queue, and hence may work its way back through the FIFO cache. This provides a large performance improvement over the FIFO algorithm, but it is dependent on a page being read as it approaches the end of the queue. Conversely, depending on the length of the queue, a page may be maintained which is not in fact required very often. The problem therefore lies in the fact that the algorithm only looks to give a page a "second chance" as it is about to be deleted from the cache. If that chance is not taken, then the page is lost, and may need to be re-loaded.

Another algorithm is the LRU algorithm. Tanenbaum describes the LRU algorithm as "excellent, but difficult to implement". The LRU algorithm approximates well an optimal page replacement algorithm. In this respect, an optimal page replacement algorithm would remove the page from RAM which is the page which will not be required again until farthest into the future (if at all), thereby putting off another page fault as long as possible. The LRU algorithm is based on the premise that pages which have been heavily used in the last few instructions will probably be heavily used again in the next few (and hence should be retained in RAM). However, pages that have not been used for a long time will probably remain unused for a long time, and hence can be replaced. Therefore, an LRU algorithm operates to try and replace the page that has been unused for the longest time.

As mentioned, whilst LRU algorithms provide a good approximation to the optimal page replacement algorithm, as acknowledged by Tanenbaum they are difficult to implement, requiring in some cases special hardware, or for software implementations, large processing overheads. It can therefore be seen that embodiments of the pagng algorithm of the present invention can be beneficial, since they can offer the benefits of LRU, but be easier to implement with smaller processing overhead.

Certain embodiments of the invention may provide a new page replacement algorithm which provides substantially the benefits of an LRU algorithm but with the implementation simplicity of a FIFO arrangement. In particular, according to the invention, pages which have been paged into memory are maintained in a paging cache, which is split into at least two parts. In some embodiments, each part of the cache is maintained as a separate FIFO, and new pages are loaded into the first part, with pages output from the first part being placed into the second part. Hence, essentially the arrangement of these embodiments is no more complex than simply maintaining two FIFO lists. However, to ensure that pages are not immediately deleted from the cache when they reach the output of the first part of the cache, they are retained in the second part of the cache, and whilst they are working their way through the second part of the cache they can be re-instated into the input of the first part of the cache whenever required. Such operation approximates a least recently used type algorithm, because whilst in the first part of the cache the page can be used by the processor. However, when the page is relegated to the second part of the cache, if it is needed again whilst in the second part of the cache it can be immediately promoted to the first part of the cache, and will not need to be re-loaded. On the other hand, if the page is not required whilst it is resident in the second part of the cache i.e. it is not recently used, then it will eventually be paged out. The second part of the cache therefore provides a buffer which acts to sort out which pages are recently used and which are not. These embodiments can therefore offer significant benefits compared with prior art arrangements.

In some embodiments, the second part of the cache is inaccessible to the processor, and the movement of the required page from the second part of the cache into the first part of the cache renders the page accessible. By making the second part of the cache inaccessible, the memory controller in charge of the process has to move the required page into the first part of the cache, and hence the page is rejuvenated, and will not be subject to relegation into the second part of the cache until it has worked through the first part of the cache. Hence, once rejuventaed, a page is available in the cache for some time.

The relative sizes of the first and second parts of the cache are maintained in accordance with one embodiment of the invention in dependence on a predetermined ratio

coefficient *R*. This provides a mechanism to ensure that the first part of the cache does not become too large, and hence that pages move through the first and second parts of the cache lists in accordance with the control algorithm. In particular, a page is removed from the first part of the cache and placed in the second part of the cache if the number of pages in the first part of the cache is greater than the product of the ratio coefficient *R* and the number of pages in the second part of the cache.

In this embodiment, a page is removed from the second part of the cache when it is necessary to free RAM for another purpose. Hence, whilst there is free RAM in the system, a page which has been relegated into the second part of the cache remains in the cache, and the cache simply grows large. This means that if a relegated page is required, it can be simply transferred back into the first part of the cache, and does not need to be re-loaded. By only reducing the size of the second part of the cache when RAM is required for another purpose, as many pages as possible can be maintained as available for transferring back into the first part of the cache, and the need to re-load pages can be kept to a minimum.

In some embodiments, when a page is accessed whilst in the first part of the cache the position of the accessed page in the first part of the cache is not altered. Hence, even though accessed, the page stays in its same position in the FIFO queue. The reason for this is to avoid too much processing overhead in maintaining the FIFO queue. Whilst in the first part of the cache the page can be accessed - it does not matter where in the FIFO queue the page is located. If the page is required again once it has come to the end of the FIFO queue in the first part of the cache then it can be re-instated to the front of the first part of the cache if it is needed again once it has been relegated. In this way, the only FIFO operations are placing pages at the start of the FIFO queue in the first cache, and no queue re-ordering is required.

There may be plural pages in the first and second parts of the paging cache, and the paging cache can be maintained at a minimum size. Maintaining the paging cache at a minimum size can help prevent a phenomenon known as “thrashing” where pages are being continually paged out from and re-loaded back into RAM. This is very detrimental to processing efficiency, as re-loading a page takes a substantial amount of time (e.g. a few milliseconds, to reload from disk).

Various modifications, including additions and deletions, will be apparent to the skilled person to provide further embodiments, any and all of which are intended to fall within the appended claims. It will be understood that any combinations of the features and examples of the described embodiments of the invention may be made within the scope of the invention.

Claims

1. A method comprising:

maintaining a paging cache of memory pages which have been paged into random access memory (RAM) for use by a processor, the paging cache being arranged in at least a first part and a second part, each part being configured to operate on a respective First-In, First-Out (FIFO) basis;

loading a new page into RAM, the new page being loaded into the first part of the cache;

outputting a page from the first part of the cache into the second part of the cache; and

when a page is required which is located in the second part of the cache, removing the required page from the second part of the cache and placing the required page into the first part of the cache.

2. A method as claimed in claim 1, wherein the second part of the cache is inaccessible to the processor, and wherein said movement of said required page from the second part of the cache into the first part of the cache renders the page accessible.

3. A method according to any of the preceding claims, wherein the relative sizes of the first and second parts of the cache are maintained in dependence on a predetermined ratio coefficient *R*.

4. A method according to claim 3, further comprising removing a page from the first part of the cache and placing the page in the second part of the cache if the number of pages in the first part of the cache is greater than the product of the ratio coefficient R and the number of pages in the second part of the cache.

5. A method according to any of the preceding claims comprising removing a page from the second part of the cache when it is necessary to free RAM for another purpose.

6. A method according to any of the preceding claims, wherein when a page is accessed whilst in the first part of the cache the position of the accessed page in the first part of the cache is not altered.

7. A method according to any of the preceding claims, wherein there are plural pages in the first and second parts of the paging cache.

8. A method according to any of the preceding claims, comprising maintaining the paging cache at a minimum size.

9. A computer program or suite of computer programs so arranged such that when executed by a computer it/they cause the computer to operate in accordance with the method of any of the preceding claims.

10. A computer readable storage medium storing a computer program or at least one of the suite of computer programs according to claim 9.

11. Apparatus comprising:

a processor; and

a paging cache of memory pages which have been paged into random access memory (RAM), the paging cache being arranged in at least a first part and a second part, each part being configured to operate on a respective First-In, First-Out (FIFO) basis;

wherein the processor is configured to cause the apparatus to perform the following:-

i) load a new page into RAM, the new page being loaded into the first part of the cache;

ii) output a page from the first part of the cache into the second part of the cache; and

iii) when a page is required which is located in the second part of the cache, remove the required page from the second part of the cache and place the required page into the first part of the cache.

12. Apparatus as claimed in claim 11, wherein the second part of the cache is inaccessible to the processor, and wherein said movement of said required page from the second part of the cache into the first part of the cache renders the page accessible.

13. Apparatus according to any of claims 11 or 12, wherein the relative sizes of the first and second parts of the cache are maintained in dependence on a predetermined ratio coefficient *R*.

14. Apparatus according to claim 13, wherein the processor is further configured to cause the apparatus to remove a page from the first part of the cache and place the page in the second part of the cache if the number of pages in the first part of the cache is greater than the product of the ratio coefficient R and the number of pages in the second part of the cache.

15. Apparatus according to any of claims 11 to 14 wherein the processor is configured to cause the apparatus to remove a page from the second part of the cache when it is necessary to free RAM for another purpose.

16. Apparatus according to any of claims 11 to 15, wherein when a page is accessed whilst in the first part of the cache the position of the accessed page in the first part of the cache is not altered.

17. Apparatus according to any of claims 11 to 16, wherein there are plural pages in the first and second parts of the paging cache.

18. Apparatus according to any of claims 11 to 17, wherein the paging cache is maintained at a minimum size.

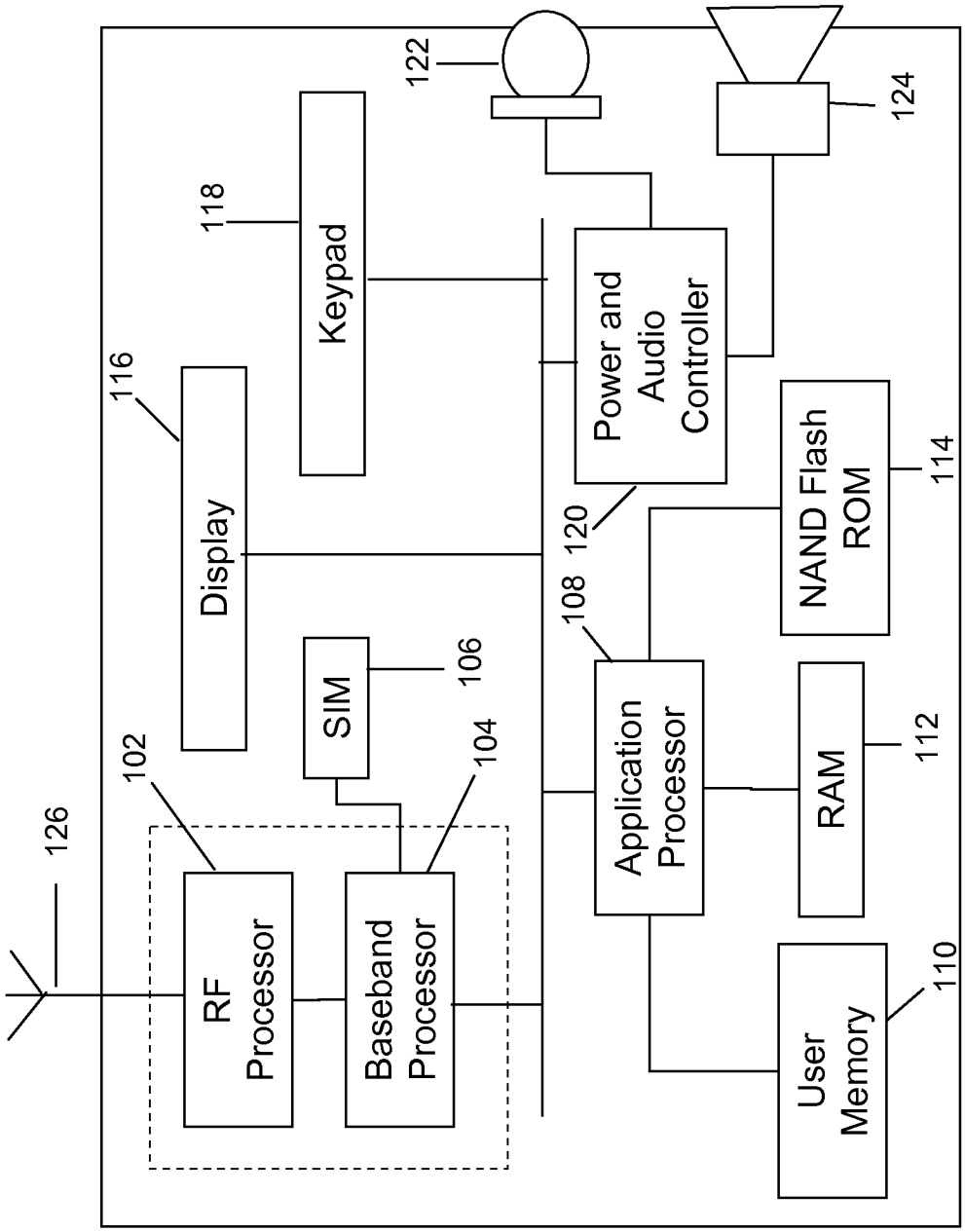

122
124
118
Keypad
Power and Audio Controller
116
Display
NAND Flash ROM
114
120
108
SIM
106
Application Processor
RAM
112
102
104
126
RF Processor
Baseband Processor
User Memory
110
10

Figure 1

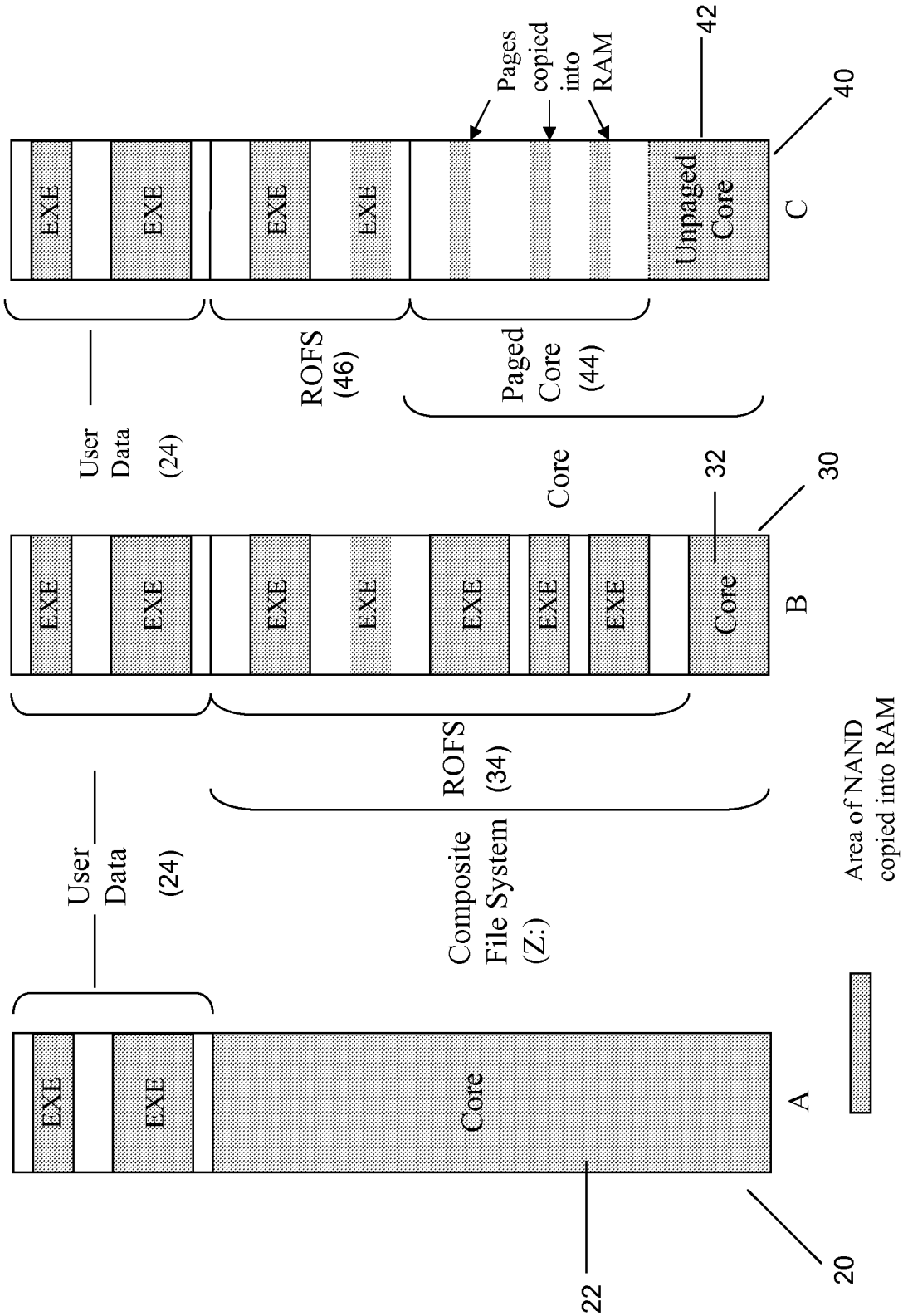

EXE
EXE
Core
A
20
22
User Data (24)
Composite File System (Z:)
ROFS (34)
Core
32
B
30
Area of NAND copied into RAM
EXE
EXE
EXE
EXE
EXE
EXE
Core
User Data (24)
ROFS (46)
Paged Core (44)
Pages copied into RAM
Unpaged Core
42
C
40

Figure 2

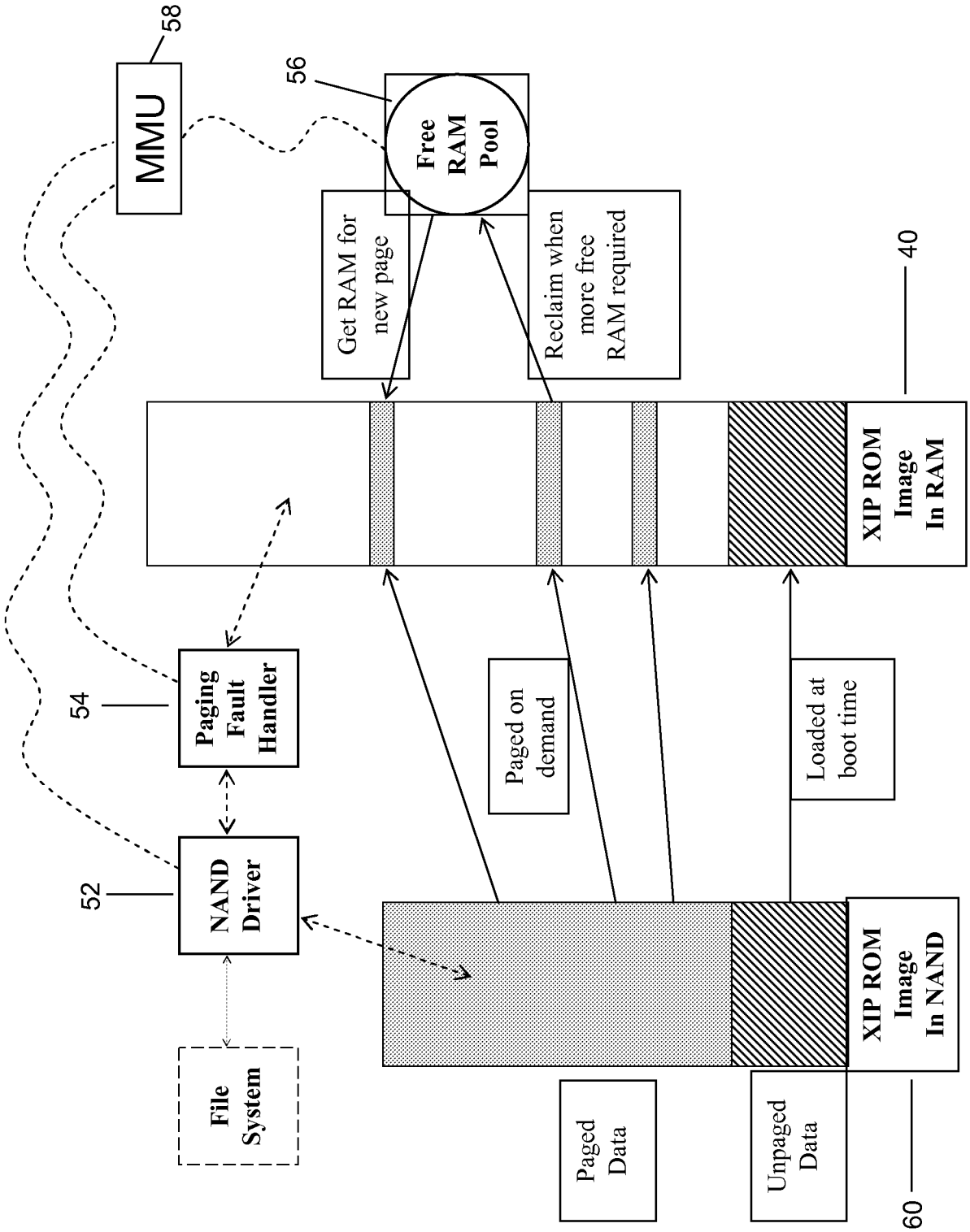
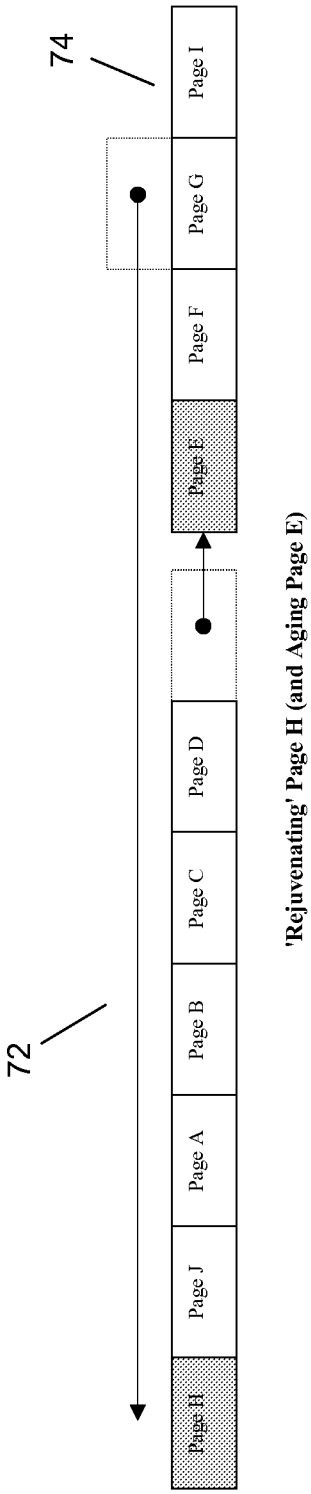

58
MMU
56
Free RAM Pool
Get RAM for new page
Reclaim when more free RAM required
40
XIP ROM Image In RAM
54
Paging Fault Handler
Paged on demand
Loaded at boot time
52
NAND Driver
File System
Paged Data
Unpaged Data
XIP ROM Image In NAND
60

Figure 3

Figure 4

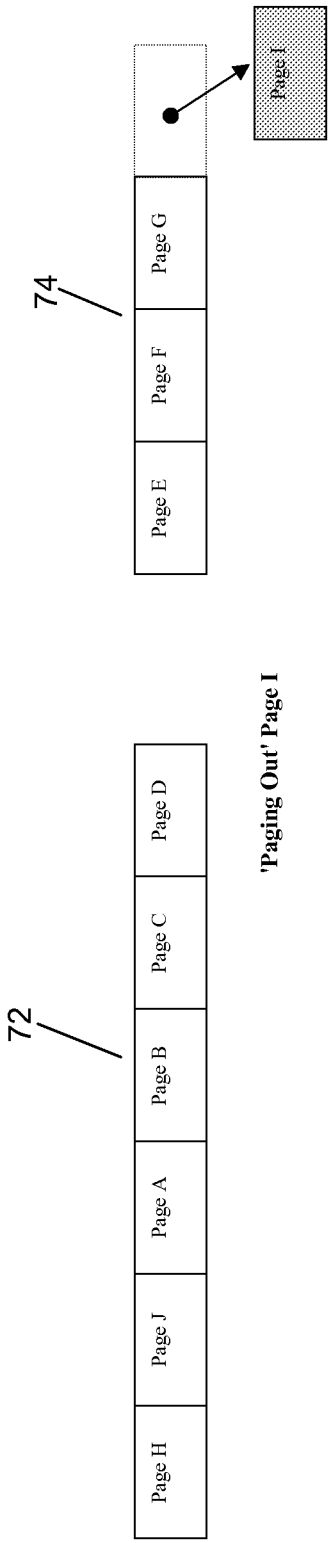
Figure 5

Figure 6



'Rejuvenating' Page H (and Aging Page E)

Figure 7



'Paging Out' Page I

Figure 8

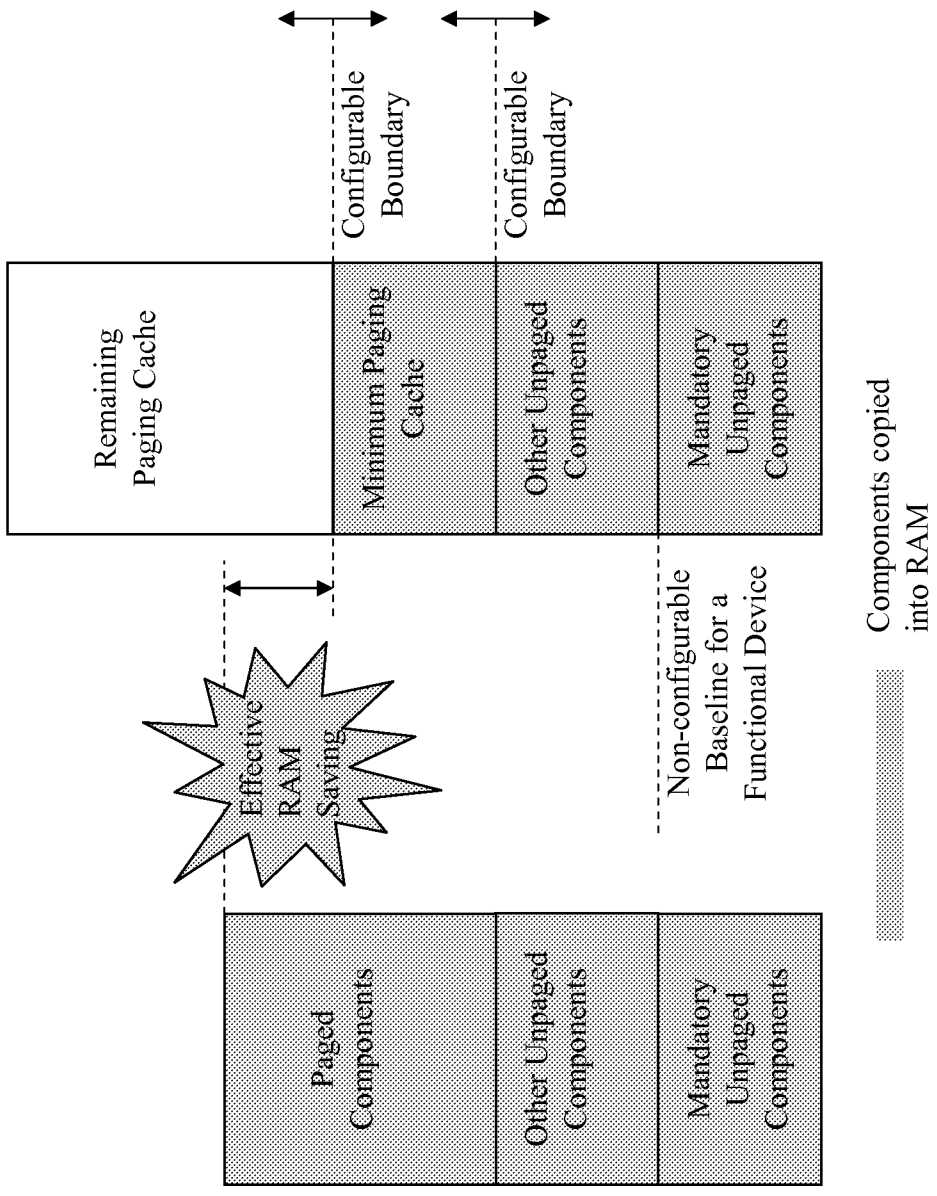

Remaining Paging Cache
Minimum Paging Cache
Other Unpaged Components
Mandatory Unpaged Components
Configurable Boundary
Configurable Boundary
Effective RAM Saving
Non-configurable Baseline for a Functional Device
Paged Components
Other Unpaged Components
Mandatory Unpaged Components
Components copied into RAM

Figure 9

INTERNATIONAL SEARCH REPORT

International application No.
PCT/FI2009/050461

A. CLASSIFICATION OF SUBJECT MATTER

See extra sheet

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

IPC: G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

FI, SE, NO, DK

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

EPO-internal, WPI, Internet

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	HANDLEY D, "Demand Paging on Symbian OS". I.Q. Magazine Online, Vol. 7, No. 2, April 2008 , pp. 71-76 [online], [retrieved on 2009-09-18]. Retrieved from the Internet <URL: http://www.iqmagazineonline.com/IQ/IQ23/pdfs/IQ23_pgs71-76.pdf>, page 73, section "Writable Data Paging" (1st paragraph); page 74, section "The Paging Cache"; section "The Paging Configuration (1st paragraph, item 2)	1-18
A	EP 1406174 A2 (MICROSOFT CORP) 07 April 2004 (07.04.2004), whole document	1-18
A	US 2007106853 A1 (EVANCHIK, S A et al.) 10 May 2007 (10.05.2007), whole document	1-18
A	US 2003110357 A1 (NGUYEN, P V et al.) 12 June 2003 (12.06.2003), whole document	1-18
A	US 5175834 A (SAWAI, W) 29 December 1992 (29.12.1992), whole document	1-18

☒ Further documents are listed in the continuation of Box C. ☒ See patent family annex.

* Special categories of cited documents:

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier application or patent but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
24 September 2009 (24.09.2009)	25 September 2009 (25.09.2009)

Name and mailing address of the ISA/FI
National Board of Patents and Registration of Finland
P.O. Box 1160, FI-00101 HELSINKI, Finland
Facsimile No. +358 9 6939 5328

Authorized officer
Olli-Pekka Piirilä
Telephone No. +358 9 6939 500

INTERNATIONAL SEARCH REPORT

International application No.
PCT/FI2009/050461

C (Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	TANENBAUM, A S, "Modern operating systems". 19.02.2008. Pearson Education, Inc., Chapter 3 "Memory management", pp. 173-252	1-18
T	SALES, J "Demand Paging on Symbian", 25.06.2009 [online], [retrieved on 2009-09-21]. Retrieved from the Internet <URL:http://www.scribd.com/doc/16775509/Demand-Paging-on-Symbian-Online-Book> especially Chapters 3 and 4	

INTERNATIONAL SEARCH REPORT
Information on patent family members

International application No.
PCT/FI2009/050461

Patent document cited in search report	Publication date	Patent family members(s)	Publication date
EP 1406174 A2	07/04/2004	KR 20040031645 A	13/04/2004
		MX PA03008766 A	10/09/2004
		RU 2348067 C2	27/02/2009
		US 2005228964 A1	13/10/2005
		US 2005235119 A1	20/10/2005
		JP 2004133934 A	30/04/2004
		BR 0304297 A	31/08/2004
		CN 1510579 A	07/07/2004
		CA 2442188 A1	04/04/2004
		AU 2003246330 A1	22/04/2004
		AU 2003243990 A1	22/04/2004
		US 2004068627 A1	08/04/2004
US 2007106853 A1	10/05/2007	US 2009150625 A1	11/06/2009
		KR 20070049062 A	10/05/2007
		CN 1963790 A	16/05/2007
US 2003110357 A1	12/06/2003	None	
US 5175834 A	29/12/1992	KR 920008441B B1	29/09/1992
		JP 2273843 A	08/11/1990

INTERNATIONAL SEARCH REPORT

International application No.
PCT/FI2009/050461

CLASSIFICATION OF SUBJECT MATTER

Int.Cl.
G06F 12/02 (2006.01)
G06F 12/08 (2006.01)
G06F 12/12 (2006.01)