VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

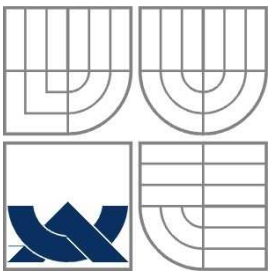
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

MEMORY MANAGEMENT IN LINUX

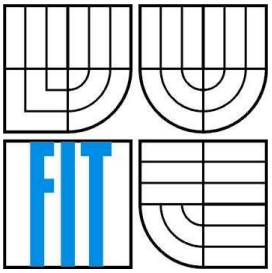
BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE Jaroslav Tuček
AUTHOR

BRNO 2007



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

SPRÁVA PAMĚTI V LINUX

LINUX VIRTUAL MEMORY MANAGER

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE — Jaroslav Tuček
AUTHOR

VEDOUCÍ PRÁCE — Ing. Tomáš Vojnar, Ph.D.
SUPERVISOR

BRNO 2007

Zadání bakalářské práce

Řešitel: Tuček Jaroslav

Obor: Informační technologie

Téma: Správa paměti v linuxu

Kategorie: Operační systémy

Pokyny:

1. Seznamte se obecně se strukturou jádra Linuxu.
2. Podrobně prostudujte koncepci a implementaci správy paměti v Linuxu řady 2.6.
3. Navrhněte a proveďte experimenty s chováním správy paměti v různých situacích.
4. Zjištěné výsledky shrňte, diskutujte a porovnejte s výsledky testů zveřejněných na Internetu a týkajících se jak Linuxu tak také příp. i jiných operačních systémů.

Literatura:

- Silberschatz, A., Galvin, P.B., Gagne, G.: Operating Systems Concepts, 6th Edition, John Wiley & Sons, 2001, 7th Edition, John Wiley & Sons, 2004.
- Gorman, M.: Understanding the Linux Virtual Memory Manager, Pearson Education, 2004.
- The Operating Systems Resource Center. http://www.nondot.org/sabre/os/articles
- The Linux Documentation Project. http://www.tldp.org

Licenční smlouva

Licenční smlouva je uložena v archívu Fakulty informačních technologií Vysokého učení technického v Brně.

Abstrakt

Práce popisuje správu paměti v jádře linuxu. První část je věnována stručnému shrnutí architektury operačních systémů a teorii správy paměti – jmenovitě virtuální paměti, stránkovacím tabulkám, algoritmům stránkování a jádrovým alokátorům. Druhá část se soustřeďuje na vlastní implementaci zmíněných principů ve skutečném operačním systému, linuxu. Součástí je též sada testů navržených pro zjištění chování paměťového správce a krátké zmínění současně existujících omezení včetně jejich navrhovaných řešení.

Klíčová slova

Operační systém, jádro, linux, správa paměti, virtuální paměť, stránkovací tabulka, algoritmus stránkování, paměťový alokátor, výkon, měření výkonu

Abstract

This work describes the memory manager subsystem of the linux kernel. The first part gives a brief account of operating systems architecture and memory management theory - of virtual memory management, page tables, page replacement algorithms and kernel allocators in particular. The second part discusses the actual implementation of these principles in a modern kernel – in linux. Finally, a series of tests stressing the memory subsystem is conducted to determine the memory manager's real behaviour. Limitations of the current linux kernel memory management and some of their proposed solutions are also discussed.

Keywords

Operating system, kernel, linux, memory management, virtual memory, page table, page replacement, memory allocation, performance, benchmark

Citace

Jaroslav Tuček: Linux Virtual Memory Manager, bakalářská práce, Brno, FIT VUT v Brně, 2007

Memory Management in Linux

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Tomáše Vojnara, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Jaroslav Tuček
22.4.2007

Table of Contents

1 Introduction

An operating system is the central software part of a usable computer system. It is responsible for managing the access to hardware and software resources of the platform, setting and enforcing policies on allocation of processor time, memory space, input and output devices and other resources to user processes. It also abstracts the low level architectural details from users, hiding away the existence of interrupts, disk blocks, physical (possibly non-contiguous) address space or competing programs loaded simultaneously in memory and provides easy to use concepts such as distinct processes, named files, virtual, contiguous, protected address spaces and a private (virtual) processor for every existing process. In this way, the kernel forms a base for other programs to use through well-defined, standardised system call routines.

Conceptually, we may divide an operating system kernel into several separate subsystems:[1]

- The scheduler responsible for handling exceptions and interrupts, system timing as well as creation, execution, switching and termination of user processes. The scheduler also sets and enforces policies on processor time sharing between runnable processes.
- Memory manager controlling the allocation and deallocation of system memory to both user processes and to the kernel itself. Management of the many kernel caches and buffers and implementation of memory sharing and memory-mapped files are also responsibilities of this component.
- Virtual file system providing an architecture independent layer over numerous physical file systems as well as the ability to represent most of the existing hardware devices as files accessible with regular system calls. Virtual file system also creates a hierarchical directory structure and allows for device-independent mounting of partitions at directory points.
- Inter-process communication subsystem providing user processes with various means of communication and synchronisation - including pipes, signals, semaphores, shared memory and message queues.

1 We are considering monolithic kernels only, other approaches to operating systems architecture all have similar functionality, albeit in different forms. For example, microkernels might have many of the mentioned subsystems moved into user space as regular processes; exokernels would go even further and keep only basic hardware allocation and protection functionality and move the rest to more conventional kernels running in its “user space” layer

- Networking support with implementations of all the protocol stacks required for participation in network communication.

This work is concerned solely with the memory manager component. First, an account of historical approaches to memory management is given - virtual memory, page tables, page eviction and memory allocation are in turn discussed in some detail. Several of these topics are further examined in the section on problems of linux memory management – in particular, the page replacement algorithms which are still subject to intensive research.

The main part of the work describes the implementation of the linux virtual memory manager, with a special emphasis on data structures used. Certain parts of the manager are omitted for the sake of brevity. These include chiefly the shared memory implementation which is more a matter of the IPC subsystem and actual workings of page outs and page ins which belong to the domain of a file system layer.

The description is concluded with a series of benchmarks measuring the memory manager's behaviour – notable among these are bandwidth, latency and scalability measurements and tests intended to determine the impact of changing certain tunable values or evaluate prospects of proposed kernel patches making modifications to the algorithms in question.

2 Memory Management

Beside processor clock cycles, memory is the most important resource in a computer system and its efficient management determines the performance (or lack thereof) of the whole platform. Chief among the memory manager's responsibilities are keeping track of free and assigned memory, its allocation on demand by user processes and by the kernel itself, deallocation when appropriate and efficient management of memory hierarchy. A good overall description of memory management can be found in [Tanenbaum01]. Here, we will present a brief account of the most important memory management topics.

2.1 Memory Management Approaches

Historically, the first memory managers made no use of memory hierarchy, they worked exclusively with main memory, there was no concept of swapping or paging and the amount of physical memory constituted the limit on the size of a runnable process. The simplest management scheme used on early mainframe computers, personal computers running MS-DOS and even today on some embedded systems allowed just the kernel[2] and one user program to be resident in memory at once. The operating system would execute the user program until its termination, then await further commands.

The transition to multiprogramming systems was first facilitated by splitting the address space into fixed-sized partitions, each able to load a different program[3]. Multiprogramming dramatically improves system throughput by keeping the CPU busy executing a different process while the original one is blocked waiting for I/O to complete; it however introduces a host of problems: memory manager must enforce more or less complicated policy decisions that determine which partition to load a new program into – for illustration: best fit implementations avoid excessive memory fragmentation, but compared to first fit algorithms discriminate interactive processes, which are usually small and tend to waste space in a partition. Moreover, programs cannot make assumptions on the memory address which they will be loaded on, thus linkers have to produce relocatable code – by providing a directory of memory addresses in a compiled program, the

2 Loaded either in a reserved part of memory or in a special ROM chip

3 IBM OS/360, for example, implemented this technique

executable can be modified before being loaded into memory by adding the starting address of the partition to every address in the code. Even more importantly, programs must not be allowed access into a partition which does not belong to it. IBM solved this problem by assigning protection bits to memory partitions and having the hardware trap an illegal access attempt. Another approach to both the relocation and protection problem was to equip the hardware with base and limit registers[4]; when executing a process, the base register was loaded with the starting address of its partition, the limit register with the partition's end. Upon every access to memory, the base register was added to the address and a trap raised if the resulting address exceeded the limit register.

With the rise of time sharing systems, these simple techniques were no longer sufficient. The first attempt to address their limitations was by swapping mechanisms. Memory managers implementing swapping divide memory into dynamically sized (and resizeable) partitions which can also be copied to backing store on a hard drive in case of insufficient amount of memory for all processes and users, thus fully utilising the entire memory hierarchy. Otherwise, swapping is very similar to previously discussed multiprogramming with fixed partitions, with the same problems of relocation, protection and fragmentation. Keeping track of allocated memory is more complex, though. Bitmaps or linked lists of holes and assigned memory areas have traditionally been used for its management.

2.2 Virtual Memory

Swapping in this form is not used by modern operating systems any more[5] and has been superseded by virtual memory architectures [Fotheringham61]. The basic idea behind virtual memory is to map virtual addresses used by processes into physical addresses of the actual chips by hardware (with operating system support), transparently to the user and on demand – without necessitating for the mappings to be contiguous or even consistent over the process lifetime. The most widely implemented technique of achieving this goal used today is paging[6].

---

4 CDC 6600 and, in a limited way (relocation without protection), Intel 8088 used this technique

5 But swapping has not disappeared entirely – on many UNIX systems, swapping out entire processes has remained as a method of load control to reduce pressure on the memory manager during thrashing

6 Segmentation, another historically popular, though not transparent, approach, used heavily in MS-DOS, Multics, OS/2 and others, adopted a different method. While paging provides protected address spaces by mapping the same virtual addresses to different physical addresses, segmentation assigns a different physical address space to each process

In paged systems, the virtual address space is divided into small[7], easily managed units called pages; similarly, physical memory is divided into page frames of the same size as pages. Processors contain a special memory management unit to translate virtual addresses of pages into physical addresses of page frames, using page tables created and managed by the operating system as mapping directories. Should a program attempt to access a page not currently present in memory, the MMU generates a page fault exception which the operating system handles by bringing the desired page into memory and restarting the faulting instruction. Thus virtual memory obviated the need for running processes to be loaded entirely in main memory, only the currently needed pages are memory resident, making optimal use of system resources and increasing the degree of multiprogramming.

There are two principal problems with paging systems. The first is that the translation must be fast as performance would dramatically deteriorate with expensive directory lookups upon every memory access. This is addressed by introducing translation look-aside buffers into the MMU. The TLBs cache recent virtual-to-physical address translations, if TLB hit rate is kept reasonably high, the system can substantially decrease the negative performance impact of paged virtual memory. Interestingly, many modern RISC architectures do not have hardware TLBs and manage the translation buffers in software. This allows for more flexible page table structure, considerably simplifies CPU design and frees die area for other purposes such as larger memory caches. However, as described in [Nagle93], software managed TLB have slower refill times impacting overall performance – kernel TLB misses contribute significantly to this effect. Recent trends in operating system architecture: shifting towards micro kernel designs, moving increasingly more functionality into user space and using virtual memory for mapping kernel data structures place further stress on TLB and decrease overall platform performance.

The second problem is the page table size. Linear, single level page tables for 32bit CPUs are probably doable (though impractical[8]) but totally infeasible for 64bit CPUs and other solutions had to be found. The most widely implemented ones are multilevel forward-mapped and reverse-mapped page tables.

7 Although nowadays, they can be very large too – 4 GB on IA64, for example

8 32bit CPUs with 4 kB pages and 4 byte page table entries would require 16 MB of memory per process (plus kernel) for page tables alone

2.3 Page Tables

Forward-mapped page tables are directories of physical addresses indexed by virtual address. Every process has its own page table; mapping is trivial, with page tables containing the respective physical address. This structure provides great flexibility, allowing easy aliasing of multiple virtual addresses into a single physical one, sharing memory between different processes, copy-on-write optimisations[9] and different protection schemes for the same memory mapped by different processes.

Multilevel page tables split this structure into a small directory, the entries of which point to actual page tables; only used page table pointers are filled and respective second level tables allocated, the unused entries remain NULL. As processes rarely access their entire address space, this technique provides the desired memory savings – extensions to more than two levels are possible, if required[10]. The downside is an increased cost of TLB misses as additional memory accesses are needed to traverse the page tables hierarchy, possibly causing further page faults and TLB misses.

Inverted, or reverse-mapped, page tables map the physical address space of the entire system into virtual address spaces of all existing processes. The physical address space being (usually) far smaller than the virtual one, very little memory is wasted; in addition, only one page table exists for the entire system. Page table entries are indexed simply by physical address (which is unfortunately wasteful should the system have holes in memory[11]), but virtual-to-physical address translation is now much more expensive as the entire page table must be scanned to find the mapping. However, efficient TLBs and hashing the entries in page tables mitigate the effects substantially: a hash anchor table is first indexed by a hash-function of a virtual address, giving a linked list of potential page table entries which can be searched quickly. Another downside is that reverse-mapped tables are far less flexible than forward-mapped solutions as all processes share the same table, protection requires involved walk-arounds and there is no easy way to implement address aliasing (global addresses are usually used instead).

[Huck93] proposes an improvement upon inverted page tables – the hashed page table combines the traditional inverted page table and a hash table into one structure, each entry of which

9 Sharing a writeable memory region in read-only mode until an actual write happens, thus avoiding the likely unnecessary allocation of private copies to each process

10 Indeed, required they are; 2.6 linux kernel, for example, makes use of four level page tables to support x86-64 architecture and even that does not map the entire 64bit address space, only 48 addressing bits are used

11 This can be a major concern with modern hardware devices like graphics adapters mapping large portions of memory for its own use

contains both virtual and physical addresses and pointers to colliding mappings, no anchor table is required. Indexing the page table by physical address is thus no longer necessary - yielding significant space advantages over traditional solutions whenever there are large unusable holes in physical memory. Importantly, aliasing can be achieved by simply adding the alias into the table, albeit at reduced hashed page table effectiveness.

2.4 Page Frame Reclamation

When free memory becomes tight, it might be necessary to evict some pages from main memory to backing storage (either to make room for pages being faulted in or to keep a minimal free memory reserve for the system[12]) - it is crucial for system performance to avoid paging out a heavily used page frame that would be faulted in soon afterwards (from a hard drive two or three orders of magnitude slower than main memory). Many algorithms for choosing a page to evict have been developed over the time, some of the more useful ones are listed below[13].

The NRU (Not Recently Used) algorithm works by having the hardware set two bits in page table entries - the referenced bit on page access and the modified bit on page write. The referenced bit is cleared in software every clock interrupt. During eviction, not referenced pages are a preferred choice to referenced ones and not modified pages to modified ones.

The FIFO (First-In, First-Out) algorithm evicts the oldest page in the system; while trivial to implement its performance is terrible as old, yet still heavily used, pages are frequently paged out. A simple improvement upon FIFO, called Second Chance, examines the pages in FIFO order, but evicts only a page with the referenced bit cleared. If the page was referenced, it is given a second chance by being moved to the tail of the examined pages with its referenced bit cleared. The move operation can be avoided by storing pages on a circular list and simply advancing a pointer to the eviction candidate, giving a Clock algorithm – a reasonably efficient solution and often used in practice.

The LRU (Least Recently Used) algorithm maintains a linked list of pages, evicting them from memory from its head and moving them to the tail upon reference. While LRU has excellent theoretical properties, modifying a linked list upon each reference makes it an unaffordably high

12 In order to avoid the unpleasant situation when there is not enough memory to even free memory

13 Although it is hard to determine absolute merit of page replacement algorithms – for example, choosing a page at random usually gives appalling performance. However, it outperforms most other solutions when the general assumption, namely that pages used often in the past will be used again, does not hold – as it does not for, say, multimedia applications

overhead solution that is rarely used. Fortunately, there is an acceptably efficient approximation to LRU - LFU (Least Frequently Used). It works by maintaining a software counter for each page and adding the referenced bit to it on each clock. Eviction affects the page with the lowest counter value as the one used on the least number of past clock cycles. Ageing can be used to avoid keeping pages that were heavily used only relatively long ago still in memory – by simply shifting the counters right on each clock, the effect of old references is progressively minimised.

The Working Set page replacement algorithm keeps track of a set of pages that a process used in a given time – a working set[14] [Denning68]. Pages to evict are chosen at random from the complement of the working set. To determine the working set, a time stamp is recorded for each page. As with NRU, hardware sets the referenced bit on access and software runs at every clock which clears the referenced bit and updates the time stamp to current time if it was set (meaning the page was accessed in this clock). With a known working set it is also possible to implement prefetching mechanisms to ensure that a process has its working set in memory right at the point of being switched to by the scheduler thus avoiding needless and frequent page faults[15].

An improvement upon the Working Set called WSClock as described in [Carr81] combines the working set algorithm with the efficiency of the clock by keeping pages in a circular linked list to avoid expensive scans – its performance and simplicity makes it a widely used solution in practice.

2.5 Memory Allocators

The memory manager's component responsible for allocating and deallocating memory is the single most important determinant of the overall system performance and consequently, its implementations are often judged above all else on a merit of speed. But kernel based allocators must also be efficient, as the amount of memory lost to fragmentation (both internal and external) and overhead is multiplied by numerous requests from the entire system. It must be well-suited for both allocations of long lifetimes (e.g. the address space of a user process) and respectively short ones (e.g.

14 Note what necessarily happens should the sum of working sets of all processes exceed the amount of physical memory: the system will be constantly paging out “working” pages and subsequently faulting them in, the resulting excessive I/O will effectively freeze all useful activity – a state known as thrashing. Load control and other thrashing prevention mechanisms will be discussed in a chapter on the problems of the linux VM

15 Though this is a mere theoretical advantage, such prefetching is not often implemented to avoid wasting scarce I/O bandwidth on reading in pages that may never be needed again

inode buffers for VFS system calls), for very large requests (e.g. a user process enlarging its heap) and small ones (e.g. any of the kernel descriptors). It should prevent leaking old data between processes, yet attempt to reuse once allocated objects. Considering these contradictory demands, kernels often implement more than one allocator. For a thorough discussion of this topic, see [Vahalia96].

The simplest solution – a resource map allocator – maintains a linked list of free memory areas to keep track of available resources. Usually, the list is sorted by starting address to allow for easy coalescing of free areas upon deallocation and a first fit algorithm is used for allocations. Though simple to implement and greatly flexible in allocation size, resource maps suffer badly from external fragmentation and their performance deteriorates significantly as the linked list grows in size. It is not used today, except for special purposes[16].

Another approach – a power-of-two free list - maintains a collection of linked lists, each of them grouping free blocks of the same size, which are all powers of two. Blocks are returned to respective lists when freed, coalescing is rarely implemented to avoid the costly linked list operations. Instead, a pool of blocks of each size is deemed sufficient and allocation requests can be blocked if the desired list is empty; alternatively, a bigger chunk of memory can be allocated to avoid blocking at the cost of excessive fragmentation. This solution is very fast, however, up to 50%[17] of system memory can be wasted due to internal fragmentation. External fragmentation can also be a problem with a lot of needlessly large pools of small blocks making memory unusable for large requests.

The binary buddy system [Knowlton65] is a considerable improvement upon the previous approach and a reasonably efficient solution often used in practice. Again, a collection of linked lists chaining free blocks is used to keep track of available resources, all block sizes are powers of two. Should a block of a desired size be unavailable during allocation, a bigger block is split in half, one half assigned to a respective list, the other one returned to the caller. Similarly, during deallocation, adjacent blocks (called mates or buddies) are merged when both free (finding buddies is very fast; as blocks always stay aligned when split, the buddy of a block of size 2^n is simply found at the block's address with the (n+1)th rightmost bit toggled). This splitting and merging is performed recursively, if possible, up to the largest defined block size; a bitmap is used to speed both operations up. With no fixed pools, memory is utilised much more efficiently and still relatively quickly.

16 System V used it to allocate kernel semaphores, linux uses a similar approach (with a bitmap instead of a linked list) to allocate memory to itself during boot time

17 Or even much more, should the non-blocking implementation be chosen

The binary buddy allocator is a popular solution in UNIX operating systems and many variations and optimisations have been proposed – some of them even abandoning the fixed binary size limitation, providing block sizes of fibonacci series members or generalised, arbitrarily sized blocks, for example. Notable among binary buddy allocator optimisations are the lazy splitting and unaggressive merging used in System V. These techniques are intended to avoid pathological splitting and merging the traditional implementations exhibit when a smaller block is allocated from a bigger one and is deallocated shortly after that. While traditional solutions would perform the splits and merges, the lazy optimisations keep the deallocated blocks unmerged on appropriate lists and avoid both the expensive merges and, potentially, repetitions of the entire operation should another allocation request of the same size be forthcoming.

Another modification of the simple power-of-two free list - the McKusick-Karels allocator, first used in BSD, keeps the block meta data off the linked lists. Thus avoiding the necessity of unfavourable rounding towards the next bigger size (and the consequent fragmentation) should the desired allocation size be itself a power of two (as is very often the case).

Mach's zone allocator came with a completely different approach. Because the cost of initialising an object often exceeds the cost of allocating its memory, the zone allocator maintains caches of initialised ready-to-use objects in linked lists; each list chaining objects of the same size is called a zone. Should a zone be emptied by allocations, another page is obtained from a lower level allocator, carved into respective objects and they in turn used to replenish the zone. Objects are returned to the zone when freed and can be easily reused. Zones can grow indefinitely and are usually purged periodically by a garbage collector.

Solaris uses a very similar approach in its slab allocator [Bonwick94]. Each type of object has its own cache as in the zone allocator, but objects support constructor and destructor procedures greatly aiding in object reuse. Caches are collections of slabs, which in turn are collections of blocks of memory obtained from a lower level allocator. This tiered architecture simplifies many operations compared with the zone allocator. Newly created objects are added to the slabs initialised by a constructor, freed objects are returned to its slab again initialised in a ready-to-use state by a destructor. Small objects are allocated directly within a page assigned to a slab including their meta data. The meta data of objects that cannot fit within a page are kept off the slab, on a special descriptor. Descriptors themselves are stored on a linked list and a hash table is maintained to provide fast object-to-descriptor mappings. The slab allocator also attempts to colour its caches – that is, to

vary the starting address of objects to improve the performance of hardware caches by decreasing the occurrence of cache line collisions.

3 Linux

Linux was originally created by Linus Torvalds while attending the University of Helsinki in 1991 as a replacement for the Minix micro kernel, written by professor Andrew S. Tanenbaum for educational purposes. Since then, licensed under GPLv2, linux has been developed and extended by the combined effort of numerous members of the open source community and made to interoperate with utilities created by the GNU project - giving rise to the GNU/Linux platform.

Once dubbed as hacker's and student's toy, linux has evolved into a competitive operating system. Combined with the cheap performance of the x86 architecture, it is quickly displacing proprietary UNIX systems running expensive RISC machines, gaining foothold in server and workstation market and making inroads into the desktop environment as well.

The current version of linux kernel, 2.6.20, offers these features:

- Full IEEE POSIX and SUS compliant unix kernel based loosely on SVR2 [Bach86], but with many improvements upon its design.
- Monolithic but largely modularised kernel architecture, allowing loading and unloading of kernel components (in many cases even during runtime and automatically on demand) or their easier replacement.
- Kernel-level support for multithreaded applications[18]. As of 2.6 version, linux is also fully preemptible, allowing for arbitrary interleaving execution flows in kernel space - a welcome feature in embedded or real-time systems.
- Linux runs on a plethora of hardware platforms, offers excellent support for symmetric multiprocessing and non-uniform memory access architectures, interoperates with many flavours of file systems, network protocol stacks and executable file types.
- The open source nature of linux ensures high code quality, low frequency of bugs and easy customisation of all components - possibly resulting in very small and compact or powerful and feature-rich systems.

18 Light-weight processes are in linux created through the non-standard clone () system call

4 Linux Virtual Memory Manager

In this section, we will give an account of the memory manager implementation in a real operating system kernel, pointing out the concepts, the rationale behind choosing them and describe the main data structures used (note that ordering of items within the structures described has not been preserved to improve readability; the source code is ordered in such a way as to avoid mapping items often heavily used together into the same cache lines [Sears00]). An excellent description of this topic can be found in [Gorman04].

4.1 Memory Organisation

Linux runs on a variety of architectures from embedded to supercomputer machines - including platforms using non-uniform memory access (NUMA[19]). Such machines have their memory divided into independent banks each intended for a specific purpose[20] and incurring different costs when accessed by different processors. Banks are called nodes in linux and are described by `struct pglist_data` structure, with the most important fields listed below[21]. Node-local allocation policy is used to allocate memory from the node closest to the requesting processor. Zones within a node are also chosen to satisfy allocations in a specific order, which is determined during zone creation and stored within its descriptor. Generally, `ZONE_HIGHMEM` is used first, sparing the important `ZONE_NORMAL`; `ZONE_DMA`, critical for hardware devices, is used only when all other zones are empty.

19 Some multiprocessor Alpha and MIPS machines, for example. But linux may use NUMA concepts to manage UMA machines with large holes in memory, regarding the contiguous, usable chunks of memory as distinct nodes

20 For example, each CPU may have its own bank of memory, access to the banks of other CPUs has much larger latencies; another bank suitable for DMA access may be located near device cards and assigned to them

21 This holds true even for UMA architectures. Linux tries to maintain as much of its concepts as possible in the architecture independent layer. Other examples of this are the four-level page tables even for architectures that do not support them or TLB handling code hooks. The architecture dependent layer resolves all conflicts – UMA machines use one statically defined node, two-level page table machines have the middle directories of zero size folding back on the global directory entry, TLB handling methods are no-ops on the many platforms that handle their TLBs in hardware and so on

```
typedef struct pglist_data {
      //The number of zones in this memory bank and their array
      int nr_zones;
      struct zone node_zones[MAX_NR_ZONES];
      //The order of zones from which to allocate memory.
      struct zonelist node_zonelists[GFP_ZONEMASK + 1];
      //A memory map of all pages for this bank
      struct page * node_mem_map;
      //Starting physical address of the node[22] and its size
      //in present and spanned pages (holes are the difference)
      unsigned long node_start_pfn;
      unsigned long node_present_pages, node_spanned_pages;
      //Node id and a this node's kswapd thread's process desc.
      int node_id;
      struct task_struct * kswapd;
} pg_data_t;
```

The before-mentioned zones are ranges of memory each suitable for a different purpose, which the nodes are divided into. With the x86 architecture, three zones are used: `ZONE_DMA` which covers the first 16 MB of available memory and its use is required by many device adapters that cannot address memory over this limit; `ZONE_NORMAL` including all the available memory between 16 – 896 MB[23] and `ZONE_HIGHMEM` covering the remainder. The difference between the last two zones lies in the way kernel maps memory. Only `ZONE_NORMAL` is permanently mapped in kernel page tables because of the limited address space of 32 bit processors[24], memory found in `ZONE_HIGHMEM` must be mapped temporarily by `kmap ()` when accessed, this mechanism will be described in detail later. Because only `ZONE_NORMAL` is permanently mapped by the kernel, the majority of operations can take place using exclusively this zone.

Consequently, it is not only the most performance-critical zone in the system, but considering that the `mem_map` array (see later in this section), page tables (though this limitation has been lifted in recent kernel revisions) and other important structures must be allocated from `ZONE_NORMAL`, it

22 The starting address has to be stored as a page frame number instead of a virtual address because certain architectures (x86 with PAE – 36 bit addressing support for 32 bit processors - enabled, for example) can address more memory than can be represented with their word size

23 The 896 MB limit is related to the way kernel and user address spaces are split. By default, 1 GB area is dedicated to the kernel, the upper 128 MB of which is reserved for `vmalloc ()` to implement non-contiguous memory allocation in a contiguous address space, `kmap ()` space used to map high memory into low memory pages and fixed mappings space required by certain subsystems that need to know its virtual addresses at compile time – such as APIC - leaving the kernel with only 896 MB of directly mapped memory

24 Consequently, 64 bit machines need neither `ZONE_HIGHMEM` nor perform temporary mappings when accessing a part of its memory, speeding memory access operations – at least in theory

places a ceiling on usable memory capacity for the system (an issue for 32bit machines with PAE, for example). Solutions exist – one possible approach is to give both the kernel and user processes separate address spaces[25]. The downside is an inevitable performance hit in the form of a TLB flush and refill per system call. Alternatively, the kernel can be assigned a bigger portion of the address space, but this may negatively influence the functionality of user space applications[26].

Zone descriptors keep track mostly of statistical data, free area information used by the buddy allocators, locks for multiprocessor synchronisation and wait tables used to queue processes waiting for I/O to complete on a desired page. Zones also determine watermarks influencing the activity of `kswapd` – the system page reclamation thread (note that there is one `kswapd` thread per each node in the system in the 2.6 kernels). `kswapd` is woken up when any zone reaches only `pages_low` free pages and does not go back to sleep until `pages_high` pages are available again. Under extreme pressure on free memory, when `page_min` free pages threshold is reached, the allocator itself will do the work of `kswapd` in a synchronous manner.

```
struct zone {
      //A lock protecting the structure from concurrent access
      spinlock_t lock;
      //The number of available pages in the zone
      unsigned long free_pages;
      //Limits which control page reclamation by kswapd
      unsigned long pages_min, pages_low, pages_high;
      //Free area bitmaps used by the buddy system allocator
      struct free_area free_area[MAX_ORDER];
      //Hash tables of wait queues of processes
      //waiting on a page
      wait_queue_head_t * wait_table;
      //These items have analogous meaning as in a zone descr.
      unsigned long zone_start_pfn;
      unsigned long spanned_pages, present_pages;
      //LRU lists, their length and a spinlock protecting them
      //See page reclamation section later in the text
      spinlock_t lru_lock;
      struct list_head active_list, inactive_list;
      unsigned long nr_active, nr_inactive;
};
```

25 http://people.redhat.com/mingo/4g-patches/

26 The obvious solution is buying a 64bit machine

The last division of memory is into pages, described by `struct page` structures which are kept in a global `mem_map` array[27]. The page descriptor keeps track of the page usage and of its belonging to respective linked lists – chaining for example all dirty pages of a memory-mapped file, all pages forming a cache in a slab allocator or all inactive pages as far as the page reclamation algorithm is concerned.

```
struct page {
      //Pages are kept on various lists through this structure
      struct list_head list;
      //The address space of the backing storage of this page
      //The structure contains call back procedures for
      //performing operations on the backing storage
      struct address_space * mapping;
      //An index within a memory-mapped file or a swap space
      pgoff_t index;
      //The reference count of this page
      atomic_t _count;
      //Pages that can be swapped out[28] are kept on an lru list
      struct list_head lru;
      //Virtual address of a page in high memory that is
      //currently mapped by kmap ()[29]
      void * virtual;
};
```

Several status flags are also kept for the page descriptor – bits indicating whether the page is active, referenced, reserved, dirty, in high memory, being swapped out and other less important flags. To save memory space, the mapping between a page and the zone it belongs to is also encoded in the status bits instead of maintaining a separate pointer. Other important mappings - between virtual and physical addresses and between addresses and their respective `struct page` descriptors – will be better understood after describing the user space / kernel space address split and page tables in linux.

Linux implements forward-mapped four-level page tables. The page table hierarchy consists of a page global directory (PGD), page upper directories (PUD), page middle directories (PMD) and page tables. Any virtual address can then be split into offsets into these tables and an offset within the actual data page frame found in the page table lookup. Beside the page frame address, page table

27 With the zones and nodes having pointers to their respective 'subarrays' of the `mem_map`

28 Technically, swapping out affects whole processes and is not used in modern operating systems; pages are paged out. But the two words are commonly used interchangeably

29 In the 2.6 kernels, this is no longer of general necessity, the field is used only if specifically required by the platform; instead, a hashtable `page_address_htable` is used to keep track of only the truly currently required mappings, saving one pointer per page worth of memory space

entries contain several protection and status flags – the self-explanatory `_PAGE_PRESENT`, `_PAGE_RW`, `_PAGE_USER` (indicates the privilege level necessary for access), `_PAGE_DIRTY` and `_PAGE_ACCESSED` flags and `_PAGE_PROTNONE` bit used to mark a page that is resident, yet inaccessible to user space, such as a page protected with `mprotect ()` system call.

Every process and the kernel has its own page table. The address space[30] is divided into a user space part and the kernel space part[31], the latter being shared by all processes in the system. As stated earlier, the kernel uses its page table to linearly map all memory in `ZONE_NORMAL` into its address space[32]. With this in place, the before mentioned mappings are trivial to implement. All processes map virtual to physical addresses using their page tables. Because kernel mappings are linear, the translation from virtual to physical address and its reverse operation are performed by simply subtracting (adding respectively) the address of user/kernel space split. When the physical address is known, determining the descriptor of the page it belongs to consists in using its page frame number[33] as an index into the global `mem_map` array of all page descriptors. The reverse operation, mapping a `struct page` to its physical address, is achieved by determining the descriptor's index in the `mem_map` array and left-shifting it appropriately.

4.2 Process Address Space

Every process in the system has its own private and protected address space – mapped to the physical address space through process page tables. The kernel never allocates memory to processes immediately, instead an area of memory with requested access permissions – called a memory region - is set aside for the process. The allocation itself is postponed until the page is actually accessed – the case of accessing a yet non-existent page belonging to a valid memory region is taken care of by the page fault exception handler, which acquires a new page from the physical memory allocator and restarts the process on the faulting instruction. Similarly, requests to copy writeable memory are postponed, respective pages marked read-only and shared between processes while assigned to a writeable memory region. Upon writing them, the page fault handler recognises such pages as copy-

30 We are considering 32bit machines alone here, the discussion does not apply to 64bit platforms without `ZONE_HIGHMEM`

31 This defaults into 3 GB / 1 GB split on the x86 architecture

32 Huge page tables (4 MB on x86) are used for the kernel page tables, if available, saving memory by avoiding one level of page tables and additionally, increasing TLB hit rate

33 Which is, naturally, determined by right-shifting the address by the number of bits in the page frame offset

on-write optimisations and allocates a new page, marking both the new one and the original as writeable. The page fault handler resolves all other cases of invalid memory references. Allocated, but not present, pages are brought into memory either from the page cache or the swap backing storage and expandable memory regions (like the stack) are grown to cover as of yet invalid space. `SIGSEGV` signal is sent to a process accessing an invalid (non-existent, non-growable) region or lacking sufficient permissions to access a valid one.

Memory regions thus group contiguous pages intended for a similar purpose – for example a process stack or a heap area, shared libraries or memory-mapped files – they are described by `struct vm_area_struct` structure, the important fields of which are:

```
struct vm_area_struct {
      //The address space descriptor of the process this memory
      //region belongs to
      struct mm_struct * vm_mm;
      //Limits of this memory region
      unsigned long vm_start, vm_end;
      //All memory regions of a process are kept on a linked
      //list and a red-black tree[34] for fast look-up
      struct vm_area_struct * vm_next;
      rb_node_t vm_rb;
      //Protection and status flags
      pgprot_t vm_page_prot;
      unsigned long vm_flags;
};
```

All memory regions for a process are kept sorted by address on a linked list for convenient sequential access (for example, when searching for a free memory hole) and on a red-black tree for fast random access (for example, when searching for a memory region covering a specific address); efficiency of random access is essential as it is required relatively often – including in exception handlers. Besides the obvious read, write and execute permissions, regions can be allowed to be shared or grown (either down – as stacks do, or up – as the heap does), memory in a region can also be locked to avoid being swapped. In case the region has a memory-mapped file backing it, the descriptor also records the respective file pointer and an offset beginning on which it is mapped.

The process address space itself is described by `struct mm_struct` structure. It keeps track of various statistical information, limits of program sections its process is executing,

34 Previous kernel versions used AVL trees which enforce more rigorous balancing to ensure better worse-case scenarios; however, AVL trees require more expensive balancing operations

synchronisation mechanisms to protect its fields from concurrent access, pointers to all of its memory regions and a PGD address. The address space descriptor of the `init` process is statically defined at compile time, all others are created as copies of the descriptor belonging to its parent process by the `fork()` system call.

```
struct mm_struct {
      //The list head and the tree root chaining memory regions
      struct vm_area_struct * mmap;
      rb_root_t mm_rb;
      //The page tables' pointer
      pgd_t * pgd;
      //The reference count of users and anonymous users[35]
      //accessing this address space
      atomic_t mm_users, mm_count;
      //A semaphore and a spinlock protecting the descriptor
      struct rw_semaphore mmap_sem;
      spinlock_t page_table_lock;
      //All address space descriptors are linked through this
      struct list_head mmlist;
      //Limits of various sections of the address space
      unsigned long start_code, end_code, start_data, end_data;
      unsigned long start_brk, brk, start_stack;
      unsigned long arg_start, arg_end, env_start, env_end;
      //Statistical data[36]
      unsigned long rss, total_vm, locked_vm;
};
```

4.3 Memory Allocators

Linux makes use of four different memory allocators. A very rudimentary bitmap based solution responsible for initialising the system during boot time, the buddy system as a general allocator of contiguous blocks, a resource map based allocator mapping non-contiguous memory into a contiguous address space and the slab allocator as a special purpose cache system for frequently used objects.

The boot memory allocator is a very simple solution. Bitmaps are used to keep track of free memory and areas suitable for allocation are searched in first-fit fashion. The allocator can merge

35 Anonymous users access only the kernel part of the address space (kernel threads, for example) – context switching to them does not necessitate a TLB flush as the page tables of the previous process can be borrowed (a technique called lazy TLB switch), greatly speeding context switch times

36 Number of resident pages (this does not include global zero page – a page assigned to the process when a new page is requested, until modified), total memory space occupied and locked pages count

subsequent allocations that do not require a whole page size, thus decreasing external fragmentation. When the kernel initialisation phase completes, the boot memory allocator retires itself. All unallocated pages[37] are given to the buddy system which from now on takes full control.

The binary buddy system is the general kernel allocator used in linux. As described previously, the binary buddy system maintains a linked lists of free memory blocks formed by a power of two consecutive pages (the powers of two range from 0 to `MAX_ORDER`[38]). The allocator searches the list of blocks of a desired size and if no block is available a bigger block is split into halves, called buddies, one of them is inserted into a proper list, the other is returned to the caller. This process is performed recursively, if necessary. Buddies are coalesced whenever possible upon being freed.

Linux does not implement any optimisations intended to avoid unnecessary splitting and subsequent merging. The increase in code complexity is probably not worth the performance increase (if any), because the caching slab allocator minimises the number of calls to the buddy system. Moreover, many parts of the kernel maintain quicklists of frequently used data structures themselves to further avoid using the potentially expensive allocator[39].

In addition, a set of caches of single free pages is maintained for each processor and zone: the hot cache and the cold cache. Pages belonging to the hot cache are likely, whereas those in the cold cache unlikely, to be still in the given processor's hardware cache. Using pages that are already cache mapped is, naturally, beneficial to system performance. But there are cases when requested pages are known to remain unreferenced for a relatively long time – for example, when performing I/O read ahead or using DMA, in case of which the processor caches are not involved anyway – then it would be a needless waste to allocate hot pages and a cold cache is used instead. Single page requests (by far the most common ones in linux) are satisfied from the cache, which is replenished when empty in one larger batch request to the allocator itself. Note that in effect, the relatively expensive splittings are deferred - achieving one of the benefits of a lazy buddy systems.

---

37 Including all pages used for data and code sections of functions called only during boot time

38 `MAX_ORDER` equals 11 in the 2.6.20 kernel

39 For example, the memory manager may maintain quicklists of page table directories (this is architecture specific, some architectures may consider caching page global directories as overzealous optimisation because they are only needed during process creation, already an expensive operation) – data are taken from these lists when needed and later returned to them when no longer so. The buddy system is only called when the quicklist in question gets empty. Also, the lists are purged when memory is tight by the `kswapd` kernel thread

Bitmaps are used to manage the state of memory blocks. To conserve memory, only one bit is used to track both buddies. Whenever either of them is allocated or freed, the respective bit in the bitmap is toggled, consequently the bit is zero if both buddies are free or both in use.

The allocator employs node-specific allocation policy to assign memory from a bank closest to the requesting processor (which, naturally, necessitates in NUMA architectures maintaining processor-ID to node-ID mappings). Zones are also tried in order determined during the node creation, which is usually such as to spare DMA memory and prefer high memory to `ZONE_NORMAL`.

The buddy system behaviour can be customised by passing several flags by the caller, the most interesting of them indicate whether the caller can sleep or perform I/O; the system can also be forced to try indefinitely in case of critical requests that absolutely must not fail. Allocations are attempted in several passes if enough memory is not immediately available, the `kswapd` kernel thread responsible for paging out unused memory is woken up between passes in that case. Should even its actions not free enough memory, the buddy system will try to free some pages itself. However, the freed memory will not be inserted into the global pool, but used to satisfy the caller exclusively.

The blocks allocated by the buddy systems are contiguous in memory. Not only is the allocation itself performed more quickly, the kernel page tables need not be modified at all, sparing the system the expense of a TLB flush. However, the buddy system suffers from external fragmentation and satisfying a request with contiguous blocks is thus not always possible. Linux provides another allocator, the `vmalloc ()`, based on resource maps, to address this issue and allocate non-contiguous memory[40].

To implement `vmalloc ()`, a part of the kernel virtual address space is reserved and its respective page tables modified by `vmalloc ()` to point to correct physical pages. The pages themselves are allocated by the buddy system. Although the kernel page tables are modified to point to the physical memory, the page fault generated by the caller upon access to an incorrect memory area is recognised by the exception handler and the page tables of the faulting process are synchronised with the reference kernel page tables. The `vmalloc ()` address space is managed by

40 This is, however, used sparingly in the kernel: module loading and swap map allocation are two principal areas where `vmalloc()` is employed

a linked list of `struct vm_struct` structures - basically, simple (starting address, allocated size) pairs.

Another part of the kernel address space is reserved for `kmap ()` to temporarily map high memory pages into low memory[41]. A similar mechanism to high memory pages mapping exists in the kernel – the bounce buffers which are responsible for performing I/O with the full range of memory available on devices unable to address it[42]. For this purpose, the I/O is performed on buffers in low memory and they are subsequently synchronised with the high memory buffer that the I/O operation caller specified. This entails an undesirable but necessary performance hit as data is copied twice during the operation.

The slab allocator is intended to offset the internal fragmentation problems with the buddy system by allowing for requests smaller than a page. Moreover, the slab allocator caches commonly used object in an initialised, ready to use state – thus compensating for the time required for initialising an object being much higher than allocating it, as is often the case. The slab allocator is made by a collection of caches chained on a linked list. Each cache is formed by blocks of page frames, called slabs, allocated from the buddy system. The slabs themselves are carved into objects that the cache manages.

To avoid the internal fragmentation problems inherent in binary buddy systems, a set of caches of objects ranging from 32 bytes to 128 kB is maintained (in pairs, one cache suitable for allocation from `ZONE_DMA`, the other from `ZONE_NORMAL`). Kernel routines may allocate memory from these buffers by calling the `kmalloc ()` function. Besides these general caches, new caches can be created with `kmem_cache_create ()` for allocation of other often used objects.

Each type of objects that is obtainable through the slab allocator has its own cache[43], described by the `kmem_cache_s` structure.

41 By default, 32 MB are reserved on x86, which may seem rather low considering the 64 GB physical memory limit of x86 processors with PAE support, but `kmap ()` mapped memory is supposed to be soon unmapped by `kunmap ()`

42 Such as 32 bit devices on 64 bit processor systems

43 The kernel exports the information on used caches through /proc/slabinfo

```
struct kmem_cache {
      //Lists (full, partial, free) linking slabs for the cache
      //are kept in this structure, along with a spinlock and
      //other required information
      struct kmem_list3 * nodelists[MAX_NUMNODES];
      //The size of objects in the cache, the size of each slab
      //in pages and the number of objects per slab
      int obj_size
      unsigned int gfporder, num;
      //Various flags indicating the state of the cache
      unsigned int flags, dflags;
      gfp_t gfpflags;
      //Per-CPU data
      struct array_cache * array[NR_CPUS];
      unsigned int batchcount;
      //Colouring of the cache for hardware optimisation
      size_t colour;
      unsigned int colour_off, colour_next;
      //Constructor and destructor functions for objects
      void (* ctor) (void *, kmem_cache *, unsigned long);
      void (* dtor) (void *, kmem_cache *, unsigned long);
      //All caches are linked through this structure
      struct list_head next;
};
```

To increase the speed of allocating an object from a cache as well as to simplify the cache reaping (that is removing free pages from the cache by the `kswapd` kernel thread when short of memory), all slabs belonging to a cache are grouped on three different lists – slabs without free objects in them, completely free slabs and partially used slabs. Allocations are always satisfied from a partially used slab, if possible.

Caches can be customised by being told how to align their objects, where to store slab descriptors (either in the slab itself or in a special cache), whether they can be subject to reaping and what kind of callers will use them (similar to the buddy system, e.g. a flag indicating whether the allocation is allowed to block the caller). The slab allocator also provides the caches with abundant debugging and statistics gathering functionality.

One of the major functions of the slab allocator is improving the performance of hardware caches and multiprocessor systems. This is achieved in two ways – by colouring the slabs and by maintaining pools of per-CPU objects in each cache. Slab colouring is a simple technique that uses memory otherwise wasted in a slab (if the slab size is not an exact multiple of the size of objects stored in it) to offset objects in different slabs of a given cache by varying amounts. Consequently, the objects would use different lines in a hardware cache and not flush themselves out.

The per-CPU pools of objects try to keep data in use on the same processor as long as possible. This is, again, beneficial by not dirtying the cache with yet unused memory addresses. Allocations and deallocations are satisfied from and to the pool; objects from the slabs will be taken only if the pool is exhausted – and in that case, in a large batch which will replenish the pool to minimise the number of calls to the allocator. Another big advantage of this technique is that spinlocks do not have to be held during requests as there is no possibility of a contention from other processors.

The slabs are described by a much simpler `struct slab_s` structure:

```
struct slab {
      //The list (free, partial, full) this slab belongs to
      struct list_head list;
      //The colouring offset calculated for the slab by a cache
      unsigned long colouroff;
      //The starting address of the first object in the slab
      void * s_mem;
      //The number of objects currently allocated from the slab
      unsigned int inuse;
      //An array used to store locations of free objects
      kmem_bufctl_t free;
};
```

To map already allocated objects to the slab and the cache they belong to, pointers within a corresponding page descriptor (those that otherwise link the descriptor on various LRU, dirty or active lists within the kernel) are used. The `kmem_bufctl_t` array then serves as a pseudo-linked list of free usable objects within a given slab.

The slab descriptors can be stored either within the actual slab or in a special cache reserved for this purpose. The desired method is chosen according to the object size. Slabs in caches of large objects[44] would suffer overly from fragmentation with slab descriptors stored within them, so the special cache is preferred.

4.4 Page Frame Reclamation

A running system is bound to use all available memory to satisfy requests by user processes, store various kernel descriptors and implement performance enhancing buffers and caches. A

44 512 bytes on x86 architecture

mechanism is therefore required for selecting page frames to be invalidated and freed in order to be used for future memory allocations. The reclamation is performed by the `kswapd` kernel thread. `kswapd` sleeps most of the time and is awoken by the buddy system allocator[45] only when `pages_low` free pages have been reached in any zone.

All pages subject to page reclamation algorithm in linux (user mode pages and pages belonging to the page cache - pages that are not free, reserved, locked, dynamically allocated by the kernel or a part of kernel mode stacks; pages assigned to some caches and the slab allocator are also handled separately) are maintained on two lists (per each zone) linked through pointers in the `page.lru` structure: the `active_list` containing the approximation[46] to a working set of all processes and the `inactive_list` chaining all reclaim candidates. When pages are first created, they are added by `lru_cache_add ()` to the `inactive_list` and get moved to the `active_list` by `mark_page_accessed ()`. Linux tries to keep the size of the `active_list` at about 2/3 of the page cache size by moving pages from the tail of `active_list` to the `inactive_list` by `refill_inactive_zone ()` function - for example, when the caches are being shrunk. The `refill_inactive_zone ()` function resembles a clock algorithm, pages at the tail of the examined list have their `PG_referenced` flag checked. If it was set, the page is moved to the head of the list with the bit cleared because it has been recently used and is likely to be used again soon. Otherwise, it is moved to the `inactive_list`.

Pages in the system may also be kept in the page cache – a collection of several caches maintained in order to decrease the number of reads from and writes to slow disk devices. These include the buffer cache of pages buffering operations with block devices and file systems; the swap cache of anonymous pages that have a slot on a backing storage assigned[47] for page-out and a cache containing pages faulted in by reading or writing a regular but memory mapped file. These pages are also kept in a hash table to be quickly located on demand. Depending on their state, all pages that have a backing storage assigned are also linked on one of three inode queues through the `page.list` field. These queues are `clean_pages` chaining up-to-date pages, `dirty_pages`

45 Though traditionally, kswapd was woken up periodically

46 Approximation because the list is not updated on every reference

47 User processes shared memory created with `shmget ()` and `shmat ()` or anonymous `mmap ()` with `MAP_SHARED` have a virtual file system attached as a backing storage – `tmpfs` or `shm` are used for this purpose

including all pages that were modified since last sync to disk and `locked_pages` of all pages currently in a locked state[48].

In earlier versions of the kernel, the swap cache's main responsibility was to group pages belonging to shared regions. This was important to implement the synchronisation between processes sharing a page, one of them having the page paged-out. Without the swap cache, should the memory be written, the process with a paged-out page would lose the update because there was no quick way to map `struct page` to all page table entries pointing to it and was consequently not attempted. Swap cache took care of this problem. The 2.6 kernels, however, implement reverse mapping, allowing for quick location of all page table entries corresponding with a given `struct page`, obviating the major need for a swap cache. The object-based reverse mapping (the objects here refer to memory regions), used in current kernels, achieves this end by maintaining a PTE-chain associated with each `struct page` – the chains are kept for memory regions, not for each page descriptor, in order to conserve memory. Memory regions of shared anonymous memory are chained on a doubly-linked list because there is rarely an exceedingly large number of such sharing processes. Memory regions of shared mapped pages, however, are kept on a priority search tree (one for each file) to improve lookup times (consider the case of `glibc` shared by almost every process in the system).

The page-out part of the reclamation subsystem takes pages off the `inactive_list` and decides how to deal with them. Locked pages are skipped, unless examined for a second time. In that case, it is better to wait for the I/O to finish and reclaim this page and the replacement algorithm goes to sleep until the I/O completes; dirty, unmapped pages are locked and scheduled for syncing to the backing storage; mapped anonymous pages have their usage counters decremented[49] and are paged out in case it reaches zero; pages not mapped by any process are either simply discarded if they existed just on the page cache, otherwise they were a part of a file mapping and are also removed from a respective inode queue.

Next, the replacement algorithm reaps caches that consist of pages not linked on the active and inactive lists – the slab allocator caches and three caches related to the file system – the `dcache`, the `icache` and the `dqcache`.

After a predetermined number of pages have been removed from the caches, user space pages are swapped out. Page tables of all processes are walked until enough pages have been freed. All

48 For example, pages that have I/O operation in execution upon them and must not be paged out

49 To determine whether the page is shared by multiple processes

pages are examined but pages either belonging to a zone that is not currently under memory pressure, on the `active_list` or the `inactive_list` and with their `PG_referenced` flag set are skipped. The page tables are walked through the list of memory regions to avoid scanning mostly sparse address spaces.

These steps are executed several times if necessary, each time with an increased priority – indicating how severely to reclaim memory. Should the reclamation algorithm fail to free enough pages, as a last resort, the system will choose a process to be killed, in hope that its pages will replenish the free memory pool allowing the original request to succeed. A victim process is chosen according to its calculated `badness` – a value that tends to be high for processes which use large amounts of memory but are still relatively young.

All swap areas in linux, either logical partitions or regular files, are described by `struct swap_info_struct` structures and linked on a list. The swap area descriptor is a rather large structure but most of its fields are of little general interest, providing various accounting functions and optimising the search for a free slot within a given area. The only interesting field is the `swap_map` array of integers managing the state of every swap slot. The array is indexed by the slot number and its values equal reference counts[50] of the respective slot. All swap areas are chained on a list sorted by priority[51].

Swap area slots are page sized blocks of the swap space. When a page is committed to be paged out, a free swap area slot is found and the page's page table entry is modified to contain the position of the found area in the swap area list and the `swap_map` index of the found slot, then marked as not present.

50 To protect against the unlikely but possible case of the reference count overflowing, the greatest possible reference count – 1 represents a permanently reserved slot

51 Swap areas on a faster disks may be given a higher priority by the system administrator and will be used first

5 Experiments

In this section, we will conduct a series of tests in order to determine the actual behaviour of a linux kernel. Unless stated otherwise, all experiments were run on a 2 GHz Athlon (Thoroughbred core) workstation with 1 GB 133 MHz DDR RAM. Kernel releases 2.6.20 and 2.4.18 (with high memory support enabled and SMP support disabled at compile time) were used for comparison.

The first of the benchmarks used to measure the virtual memory subsystem performance is lmbench[52] [McVoy96], a suit of programs designed to uncover bottlenecks in performance of a wide range of applications. The part of lmbench that we are interested in stresses the system by a series of small latency and bandwidth critical loads moving data among the processor, cache, memory and disk drive – determining not only the performance of the underlying hardware platform but also any software limits imposed by the operating system.

lmbench's memory bandwidth benchmark measures the system's ability to copy, read and write data of varying sizes, later versions of lmbench also include McCalpin's STREAM and STREAM version 2 benchmark tests [McCalpin95].

Copy bandwidth is determined in two ways: first by a user-level library `bcopy ()` interface; second by a hand-unrolled loop that loads and stores memory-aligned words[53]. The tests vary the size of memory blocks copied - effectively bypassing processor hardware caches for sufficient sizes; care is also taken for the source and destination memory addresses not to map into the same cache line[54]. The copy test works with bytes copied, not moved - thus the results should theoretically be at best half (or third in case of less advanced architectures which perform another read before write of memory about to be overwritten[55]) the values of the read test or the McCalpin's stream benchmark.

Memory reading is measured by an unrolled loop that sums up an integer series stored sequentially in memory. An optimising compiler is highly desirable in this benchmark as to avoid generating too many assembly instructions and placing the bottleneck on the processor -

52 http://www.bitmover.com/lmbench/

53 Or 4 bytes on x86

54 The source and destination memory blocks are originally allocated with the `valloc ()` function and consequently, page aligned. Finally, one of the buffers' beginning address is advanced by a predetermined number of bytes

55 For example in order to maintain cache coherency in multiprocessor systems

consequently, the addition instruction is required in order for the reads not to be optimised away as redundant. Considering the relative speeds of today's processors and memory subsystems, the overhead of one integer addition is negligible. Memory writing is measured by very similar means and will not be discussed further.

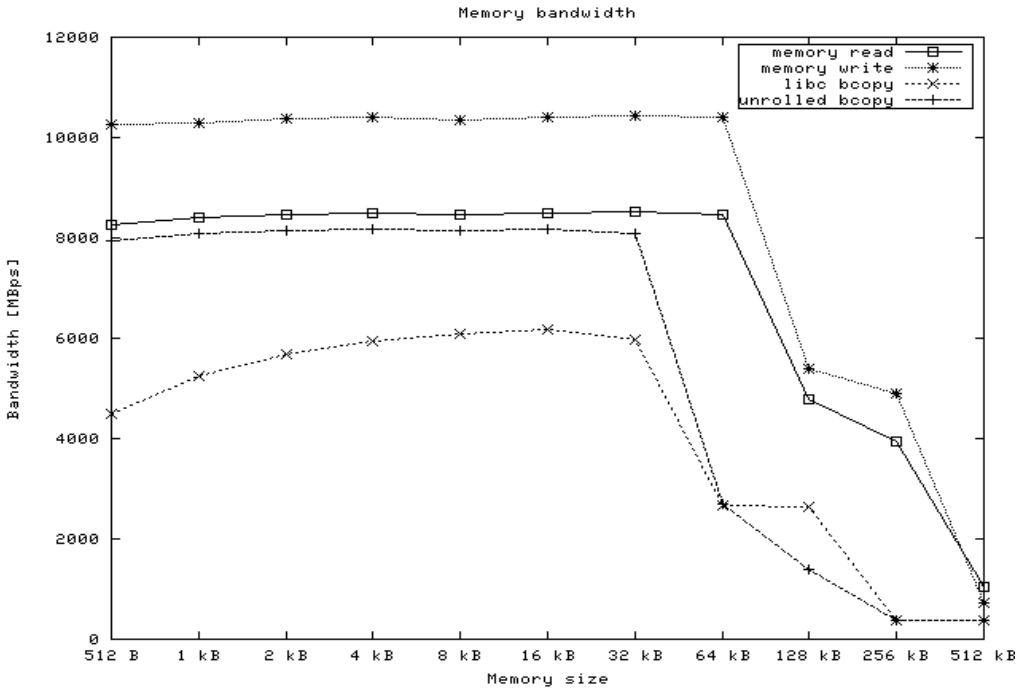


Illustration 1: Various memory bandwidths. This particular machine has 256 kB L2 cache. As an example, exact values for the 512 kB memory size are 944 MBps memory read bandwidth and 385 MBps bcopy () bandwidth - somewhat under the 1/2 theoretical limit for the x86 platform

Another area of interest, as far as memory management is concerned, is lmbench's cached I/O benchmark set intended to test the efficiency of reusing data in the file system page cache through the `mmap ()` system call. Note that no I/O is performed during these tests, the file about to be mapped is first copied into a private temporary version which effectively results in the file being forced into the page cache; `mmap ()` then maps the entire file to the process address space as a distinct memory region and reads it as any regular memory block. Good systems will have `mmap ()` results approaching the results of the memory read test - because the file system overhead is virtually zero - but operating systems in general (and linux in particular - as found in [McVoy96]) have traditionally performed dramatically worse.

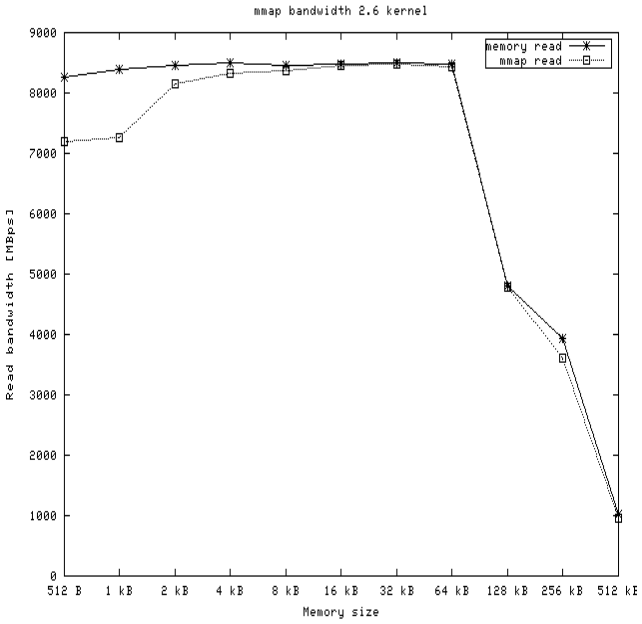


Illustration 2: 2.6 kernel mmap () bandwidth

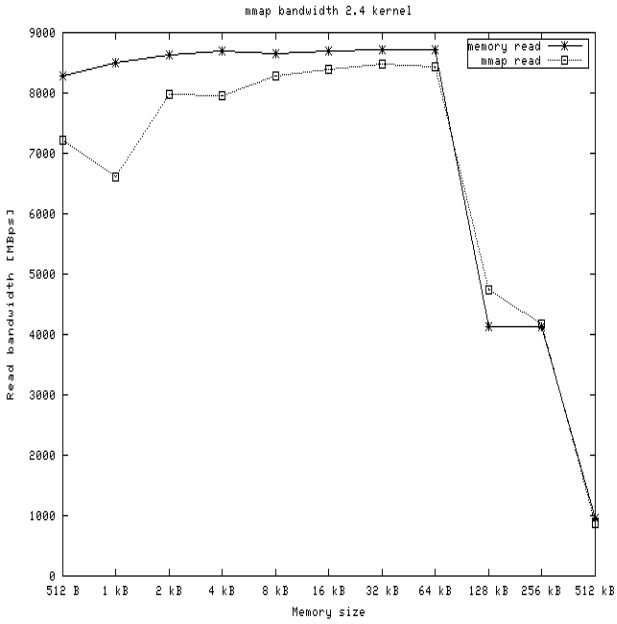


Illustration 3: 2.4 kernel mmap () bandwidth

We can see that both 2.4 and 2.6 kernels perform very well in this respect, 2.6.20 having nearly identical memory read and `mmap ()` bandwidths.

Memory latency measurements reflect not only the performance of the underlying hardware architecture but also the efficiency of prefetching algorithms, both hardware and operating systems based. lmbench measures the back-to-back-load memory read latency, which is the time each cache missing load takes, assuming the instructions before and after are also cache missing loads. If desired, the entire memory hierarchy can be measured, including the latencies and sizes of various processor caches, main memory and (possibly even) TLB miss impact [Saavedra92] by varying the array size and stride during testing. For each size, a list of pointers is created for all of the different strides, loads equivalent to C code `p = * p;` are executed and their time reported. It is assumed that the processor is capable of executing a load instruction in one clock cycle and its length is subtracted from the measured time, thus yielding pure latency. Plotted results display the entire memory hierarchy – multiple levels of on- and off-die caches and the main memory.

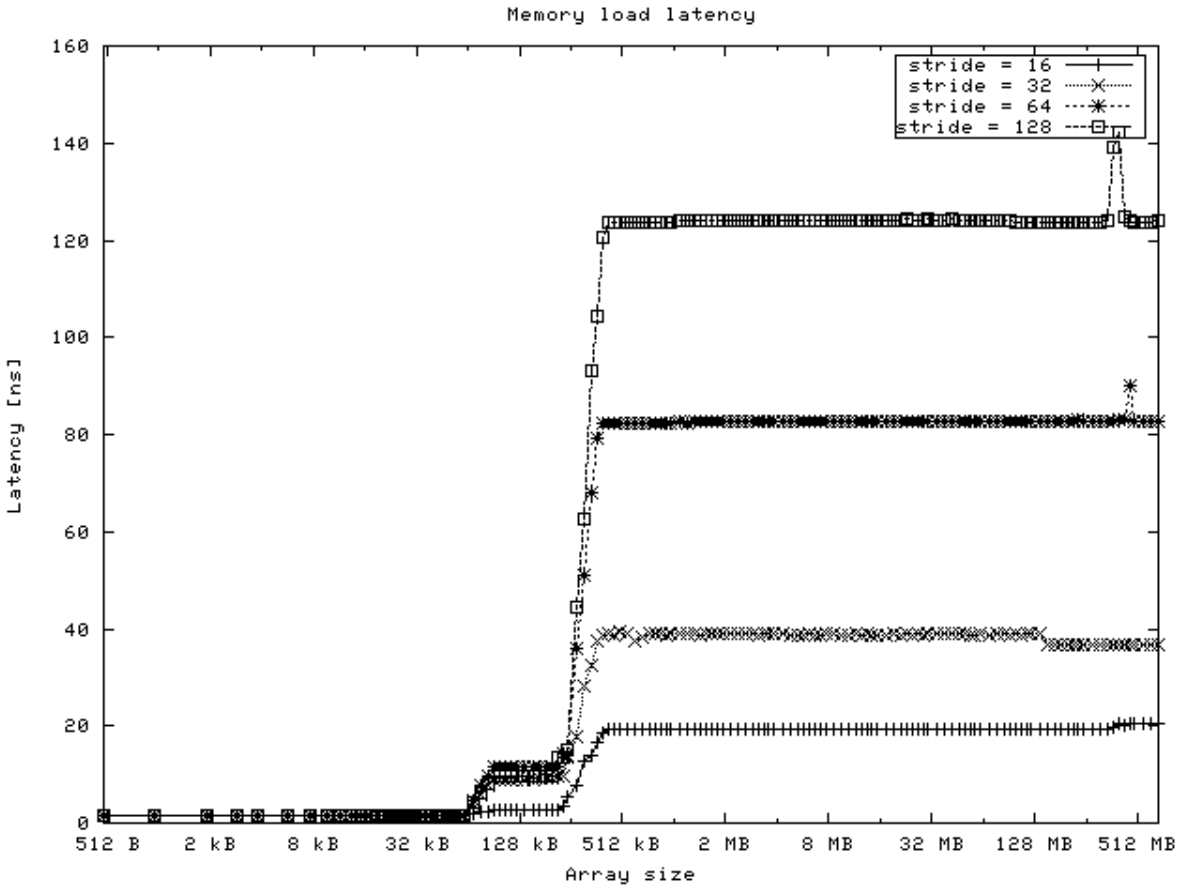


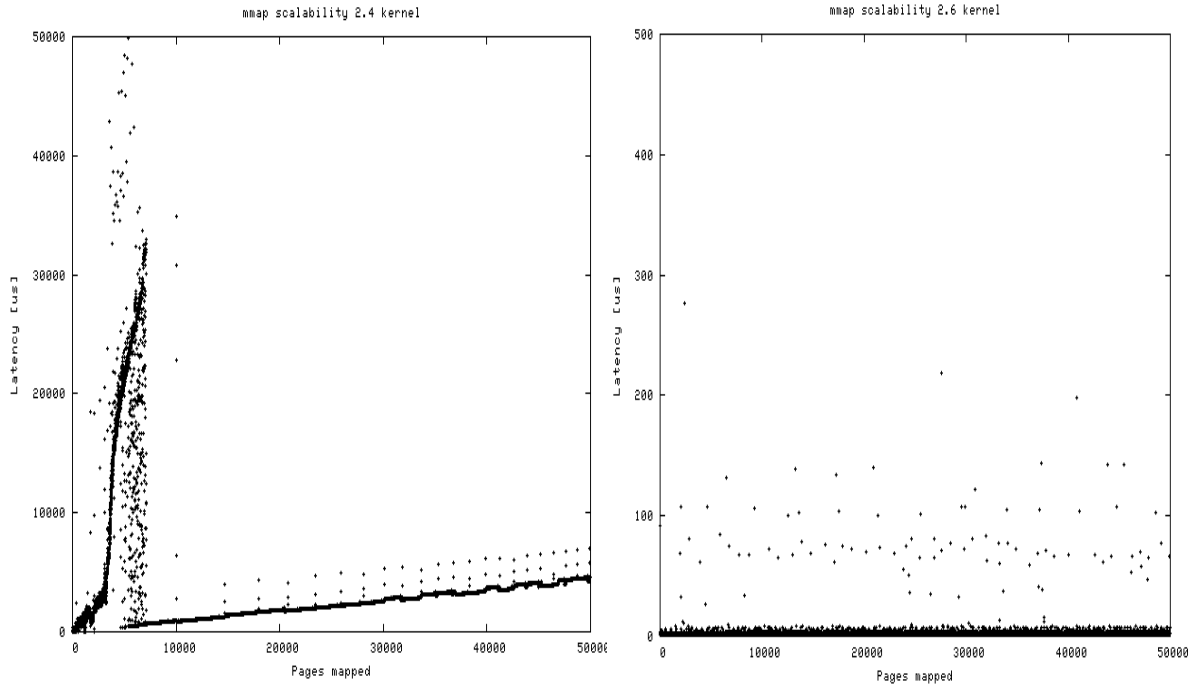
Illustration 4: Memory latencies. By observing the latency plateaus on the graph, we may determine that this particular machine has 64 kB level one and 256 kB level two on-die caches.

Next, we will explore the scalability potential of the linux memory manager with a simple `mmap()` benchmark inspired by the gatling[56] experimental web server performance analysis suite. The benchmark consists of mapping a large number of files into memory. In order to avoid the need to create many unique files, we will `mmap()` distinct page-sized chunks of a single large file (thus creating a large number of memory regions[57]) and measure the latencies of the operations. Performance-wise, this may be particularly important for object oriented database management systems or network servers which are required to handle request for many files at once. We will try to determine whether the operating system overhead increases as the number of memory regions belonging to our process rises. To make sure the file system does not become a bottleneck during

56 You may get gatling through anonymous cvs here: cvs -d :pserver:cvs@cvs.fefe.de:/cvs -z9 co gatling

57 A list of memory regions for each process is available in /proc/{$PID}/maps

testing, the benchmark process will start by reading the first byte of every page to force them into the page cache.



The results are truly impressive for the 2.6 kernels – they scale with O(1) with negligible latency for most of the system calls – a large improvement upon the O(n) linear scaling of the 2.4 kernels with latencies in thousands microseconds.

These improvements are mainly due to two changes between the 2.4 a 2.6 kernel revisions. First, process descriptors were made to cache the first available hole in their address space to improve search times; this is very important for processes with a large number of memory regions, because finding a free hole cannot be achieved by using a tree and involves a linear walk through a linked list. Second, finding pages in the page cache no longer involves linear searches. In the 2.6 kernels, pages are kept on a radix tree instead, greatly improving performance. Incidentally, though this is not the case in our benchmark, the 2.6 kernel can also perform non-linear virtual memory areas population if passed the `MAP_POPULATE` flag to the `mmap ()` system call. This would cause the system to populate page tables for a file mapping by performing read-ahead on a file (in our benchmark, the file is read entirely into the page cache beforehand, so there are no major page faults accessing it anyway).

Concerning the 2.4 kernels, you can see a sudden latency jump around 5,000 mapped regions. Obviously, the kernel must be detecting an excessive pressure on this part of the memory manager

and switching to an algorithm or data structure more suitable for the demand. The exact nature of the optimisation, however, I have failed to determine.

5.1 Tunables

The behaviour of a running kernel can be customised and controlled through the `sysctl (8)` mechanism or by writing desired values directly into respective files located on the `/proc/sys` file system which represent configurable kernel parameters. Of especial interest to the memory manager are the files found in /proc/sys/vm directory, each of them (available in the 2.6.20 kernel release) will be briefly discussed below (most are documented in Documentation/sysctl/vm.txt in the linux source code directory).

`block_dump` parameter turns on and off the block I/O debugging, which, when enabled, causes the kernel to report all read and write operations and any block dirtying of pages with a file backing storage attached

`dirty_background_ratio` parameter configures the percentage of total system memory that will, when dirtied, trigger the background write-back by the `pdflush`[58] kernel thread

`dirty_expire_centisecs` parameter defines when dirty blocks qualify as old and consequently subject to writeback

`dirty_ratio` parameter defines the percentage of system memory at which dirty writeback will be performed by the generating process in synchronous fashion

`dirty_writeback_centisecs` parameter defines whether the `pdflush` daemon should be woken up periodically, and if so, how often

`drop_caches` parameter will cause, when written, the kernel to immediately drop the contents of the page cache, the dentry cache and the inode cache in order to free their memory

`laptop_mode` parameter causes the kernel to flush all dirty blocks during any physical disk I/O thus avoiding unnecessary hard drive spin ups in the future and conserving battery capacity

`legacy_va_layout` parameter disables the 32-bit `mmap ()` map layout introduced in the 2.6 kernels and makes the kernel use the legacy 2.4 layout for all processes

58 `pdfush` has replaced the functionality of `bdflush` (which scanned the page cache looking for dirty pages) and `kupdate` (which ensured that no page would remain dirty for too long in protection from data loss in case of power failures) kernel threads used in earlier kernels

`lowmem_reserve_ratio` is the user-definable low memory watermark for each memory zone mentioned earlier which triggers the awakening of the `kswapd` kernel thread

`max_map_count` parameter defines the maximum allowable number of memory regions per process (defaults to 65,536 which can be limiting for certain kinds of applications – object oriented database systems, as mentioned in the `mmap ()` benchmark section, or `malloc ()` debuggers, which may need to create up to two memory regions per allocation)

`min_free_kbytes` specifies the memory reserve which the kernel dips into only while allocating for high priority requests when low on free memory – mainly used for serving non-blocking requests (issued mostly by interrupt handlers)

`nr_pdflush_threads` is a read-only value indicating the count of concurrently running `pdflush` threads

`overcommit_memory` controls the memory overcommitment kernel feature. When disabled, the total address space commit of the system is determined as a sum of the swap area space and a configurable part (see next parameter) of the system physical memory. When enabled, it allows processes to allocate (but, naturally, not use) more memory than available – two modes of operation are possible in this case – always overcommit and the heuristic default. The heuristic allows slight overcommits while trying to block overgreedy attempts to hog memory (this restriction is somewhat looser for super-user processes)

`overcommit_ratio` parameter defines which part of the physical memory will figure in overcommit calculations

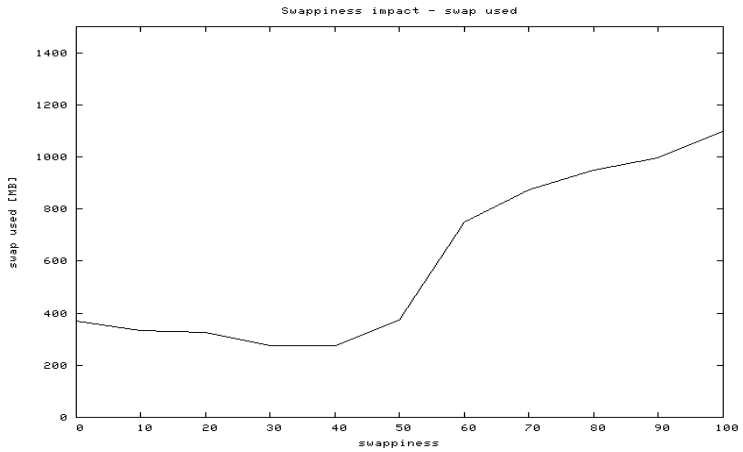
`page_cluster` is a binary logarithm value of a page-sized cluster which is written to swap in a single operation, in other words the swap I/O size. Defaults to 3, meaning 8 pages (32 kB on x86) worth of data

`panic_on_oom` parameter determines whether the kernel should panic or invoke an out-of-memory killer feature when running out of free memory. The out-of-memory killer is a somewhat controversial part of the kernel as a martyr process is determined by comparing the badness ratios of all running processes. A process with the highest value is killed. Badness is calculated in such a way as to penalise young processes with large amounts of allocated memory, but less than wise guesses do happen

`percpu_pagelist_fraction` parameter determines a fraction of pages per zone that are allocated for each per-cpu, per-zone hot cache of single pages (see the description of buddy allocator optimisations for details)

`swappiness` influences what ratio does the kernel prefer dropping pages from system buffers and caches to swapping out memory belonging to processes. Generally, setting this value lower will tend to improve interactive response (as processes are more likely to be kept in memory) while higher values will benefit system throughput (as memory is given to more immediately useful buffers instead of being wasted by processes that need not use it again at all)

We will demonstrate the impact of swappiness on system behaviour by a simple benchmark. First, we will launch a process which calls `malloc ()` to allocate a large portion (90% or so) of memory (touching every page to force the kernel to perform the actual allocation), then goes to sleep; Subsequently, we will use `dd` to simultaneously copy 4 GB worth of data from `/dev/zero` to two files located on separate hard drives (swap partition is located on a disk of its own)[59]. We are interested in determining how much of the memory grabbed by the sleeping process will be swapped out to make room for buffers improving system throughput.



The results show that increasing swappiness does not make much of a difference (that is benchmark specific, of course) until about the value 50, when the system starts to page out process memory quite heavily. Most of the `malloc ()` space is paged out by the value 70 as the system uses its memory for buffers and caches.

59 The tests are repeated for different values of swappiness, `echo 1 > /proc/sys/vm/drop_caches && swapoff -a && swappon -a` are used between the runs to ensure the results are not affected by previous activity

Swappiness is a matter of some controversy among kernel developers. Andrew Morton, for example, proclaims to have all his computers set at swappiness 100 as not to prevent the kernel from using available memory for something useful. On the other hand, Rik van Riel adamantly pushes swappiness to 0 for interactivity on desktop computers[60]. The kernel defaults to 60.

What kind of performance increase the enlarged buffers can bring depends on the degree of disk data reuse by the application. We will demonstrate the effects of the extreme values, 0 and 100 respectively, on the performance of a script which copies the same 400 MB file ten time in sequence to `/dev/null` (simulating the behaviour of a high reuse program) with 995 MB total out of 1024 MB system memory held by sleeping applications. Ideally, only the first copy should perform disk I/O, the rest should go from the page cache. But this is obviously impossible without the memory hogging application, at least partially, paged out.

Swappiness	**Average copy bandwidth**
0	27.60 MBps
100	133.86 MBps

`vdso_enabled` parameter triggers the creation of virtual dynamic shared objects for processes (enabled by default). When enabled, a page with such an object, called the `vsyscall` page, is mapped into process address space and passed to `glibc` upon `exec ()`. Its purpose is to speed up system calls made by the process by providing an optimal method of entering kernel space. Processes can take advantage of this capability by using `call 0xFFFFF000` instead of the traditional `int 0x80` for initiating system calls

`vfs_cache_pressure` parameter controls the kernel's tendency to reap the dcache and inode cache compared to the swap and page caches

60 http://kerneltrap.org/node/3000

6 Problems

This section will list the major known shortcomings of contemporary virtual memory managers, with an emphasis on the problems and limitation of the implementation found in the linux kernel.

Linux and most, if not all, other current operating systems implement page replacement algorithms that try to keep recently used pages memory resident, with the assumption that such pages are probable to be used again soon. However, this assumption is no longer valid for an increasing number of today's typical workloads and applications. For example, garbage collection systems do not explicitly free memory which they are not going to use again and may not reuse memory quickly; moreover, the garbage collector itself often has access patterns completely different from the program that uses its services. Streaming I/O, such as multimedia or data mining applications, will likely never access a recently used page again; pages these kinds of applications need, the not (for some time) accessed ones, are pages the traditional eviction algorithms are designed to page out. Many advanced, adaptive, algorithms have been developed to cope better with the situations when traditional solutions fail, their one common characteristic is the need to keep track of past memory usage pattern. We will describe some of the most promising ones here[61].

ARC, the Adaptive Replacement Cache [Megiddo03], tries to achieve dynamic, on-the-fly, adaptation to varying system workloads, without any a priori tuning of the algorithm parameters. ARC maintains two lists of pages, one chaining pages that have been accessed just once in a given time period (cold pages); the other contains pages that have been accessed at least twice in the same period (hot pages). Consequently, pages on the former list can be thought of as belonging to process parts exhibiting recency-reuse behaviour, while the pages on the latter list exhibiting frequency-reuse behaviour. The relative size of the lists is modified at run-time according to the actual workload – the list experiencing hits is grown at the expense of the other list – this learning method ensures continual adaptation to varying conditions in the system. There are many areas where the caching algorithm may be used; when applied to operating systems, a variation can be adopted, which keeps both lists at roughly the size of physical memory (so their combined size equals twice the system memory) and

61 Unofficial experimental patches to the mainline linux kernel exist that implement all the listed solutions

the learning mechanism continually varies the ratio of each list that would actually remain in memory, the remainder is paged out to backing store.

LIRS, the Low Inter-reference Recency Set [Jiang02], attempts to address the limitation of LRU-like algorithms (making eviction decisions solely on the basis of recency) by keeping an inter-reference recency (IRR) counter for every page. The IRR records the number of pages that have been accessed between the last and the pen-ultimate access to the give page. Pages with the highest value of their IRR are the current eviction candidates. In this way, LIRS avoids the problems sudden bursts of references cause to LRU (e.g. sequential scans of large files forcing the page-out of still heavily used data), because the pages accessed just once may have a very low recency but their IRR is effectively infinite as there was no pen-ultimate access to their data.

Both LIRS and ARC were originally intended for I/O cache management and their implementation in a general purpose OS memory manager entails a relatively high overhead cost. Clock-Pro [Jiang05] attempts to combine the features and performance of LIRS with the simplicity of the LRU clock algorithm. In Clock-Pro, as in LIRS, the inter-reference recency is used to determine the replacement candidate. Pages with large IRR are called cold, pages with low IRR hot (we may think of these set as the `inactive_list` and `active_list` in linux); cold pages are given a test period, during which if accessed, they are marked as hot. Resident cold pages are the reclaim candidates. All possibly reclaimable pages in a system are placed on a circular linked list and three hands move around it. The hot hand points to a hot page which has been unused the longest; the cold hand points to the longest unused cold page and the test hand points to the last cold page in the test period.

The search for a page to evict starts at the cold hand position. The page pointed to is evicted if it has its referenced bit unset. Otherwise, the cold hand continues advancing until an unreferenced cold page is found and reclaimed. A page is spared if its referenced bit is set (it is reset by the hand); moreover, if the page is in the test period, it is marked as hot, because, in effect, an access in the test period can be thought of as a low IRR. This triggers the movement of the hot hand – the hand advances (resetting the referenced bits of hot pages in the process) until it finds an unreferenced hot page, a hot page with the currently largest IRR, and marks it cold. When it encounters a cold page, it performs the same work as the test hand (which is advanced only when the number of non-resident pages reaches certain limit) – terminates their test period.

To better illustrate the promise of page replacement algorithms based on more information than recency, we will perform two benchmarks. First, a simple test consisting of allocating an array slightly larger than the available pool of memory and then repeatedly walking it sequentially without reusing the data in between the subsequent walks – a common scenario of most recently accessed pages evicting pages that will be needed the soonest. The results are for kernel revision 2.6.18 with and without Peter Zijlstra's Clock-Pro patch[62] applied booted with 96 MB of memory; an array of 100 MB is walked 100 times in strides of a page size (4096 bytes).

The second test, based on an example from [O'Neil93], randomly accesses a database through a B-tree indexed key. We will use miniDB[63], a barebones database management system, to create an approximately 700 MB database file with an index file about 1/10 of its size. Obviously, it would be desirable to keep the entire index file memory-bound because any of its blocks are accessed with much higher probability than the data file blocks are. There will be many more accesses to the data file, though, and a page replacement algorithm based solely on recency will happily evict the index pages to make room for data pages that are extremely unlikely to be needed again any time soon.

Kernel	**Sequential Scan**	**Indexed Database**
2.6.18 [Vanilla]	27m:13s	2h:06m:16s
2.6.18 [Clock-Pro patched]	13m:06s	2h:03m:31s

In theory, both tests should benefit hugely from the properties of Clock-Pro and we can see that it more than doubles the performance of our sequential scan benchmark[64]; the improvements to our database application are more modest, though - with Clock-Pro achieving approximately a 2.2% time decrease.

Even with an optimal page replacement algorithm, paging would sooner or later occur – and with the ever increasing gap between memory and hard drive latencies, its costs continue to rise. With a high enough load, any system can be brought to the point of thrashing when all useful computation virtually stops as processes spend most of the time waiting for I/O to complete instead of computing. Consequently, there is a need for a mechanism of keeping the pressure on the memory manager

62 http://programming.kicks-ass.net/kernel-patches/page-replace/2.6.18-pr1.patch

63 http://master.kernel.org/~marcelo/benchmarks/mdb-bench-2.1.tar.gz

64 This property of eviction algorithms is called scan-resistance. LRU-like algorithms do not possess it

within reasonable bounds. Traditional solutions of thrashing prevention – like temporarily suspending or even swapping out entire processes – necessitate in more complex, multi-level schedulers which have to consider not only fair CPU sharing but fair memory residence as well, because load control should not penalize any process exceedingly compared to the rest of the system; every individual process must be guaranteed to make eventual progress. Combined with other requirements for load control mechanisms – e.g. self tuning ability and preferential treatment of interactive processes[65] - the traditional methods are not satisfactory.

Linux implements (since 2.6.11 version) another layer of thrashing prevention - a swap token tuning method of load control [Jiang05a]. A token is introduced into the system that is passed to a selected process during a prethrashing phase – after the algorithm has determined that thrashing is forthcoming but well before the system detects a high enough pressure on the virtual memory to start suspending processes. The ownership of the token gives the process immunity from page out, allowing it to quickly establish its working set. It is hoped that the load spike the system experiences is temporary and can be overcome in this way – by allowing select processes to quickly progress, eventually reducing the overall load without having to swap out a single process. There are still problems with the swap token passing implementation, however. For example, although the algorithm provides considerable benefits during high loads, it is detrimental to system performance under very light virtual memory pressure. Also, ensuring fair passing of the token between processes is not completely solved as of now.

We will demonstrate what kind of performance to expect from the swap token tuning by a benchmark that forks off ten processes, each of them allocates and uses a chunk of memory for a predetermined time period and then terminates. The results were obtained on a workstation with 768 MB memory, ten processes each of which allocated 130 MB and read 2 bytes of every page 35 times. 2.4.18 kernel was used as a reference system without any kind of thrashing prevention. 2.6.18 kernel with swap token tuning enabled and `swap_token_timeout` (more on this later) set to 300 represented the load control algorithm.

Kernel	**Time required**
2.4.18 [No thrashing prevention]	12m33s
2.6.18 [Swap token tuning]	5.1s

65 A shell being used by the system administrator to solve the current overload situation should definitely not be swapped out

The algorithm can be parametrised by one newly introduced `sysctl ()` variable – `swap_token_timeout`. It specifies the length of the period a process is granted the swap token for. The value is in units of `HZ`[66] and defaults to 300, which may not be optimal as our test[67] shows.

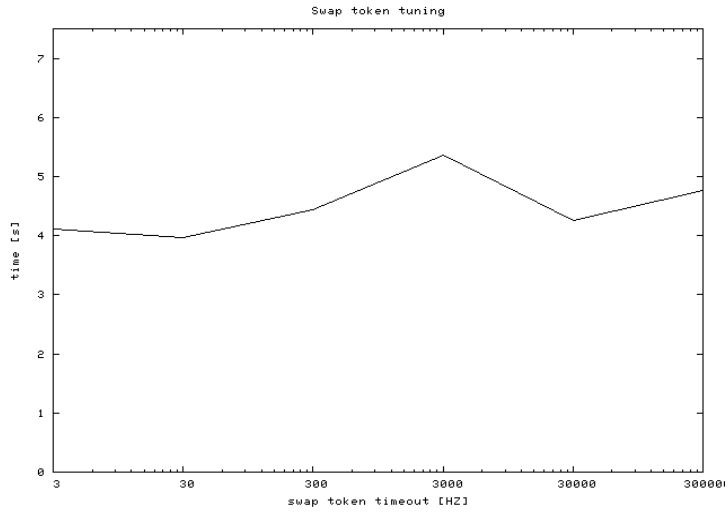


Illustration 5: The same swap token tuning test as in a previous section (1 GB workstation and 10 processes allocating 150 MB each) is now performed for different values of `swap_token_timeout` - the optimal value seems to be 30

The last issue connected with page replacement that we will mention here is the possibility of swap prefetching. With any global[68] page replacement algorithm, a large (memory footprint-wise) application is bound to cause page evictions from memory owned by other processes. When the application exits, the system is left with a large pool of free memory while considerable portions of other processes are paged out leading to poor interactivity when they are switched to by the user again.

A number of attempts have been made to counter this undesirable effect but no solution that would not negatively impact the overall system performance has been found so far. The implementation merged into the -mm source tree[69] (since version 2.6.16) tries to keep the overhead and negative influences to a minimum. A new, low-priority, kernel thread is introduced to perform

66 `HZ` is a kernel macro which equals the frequency of the timer interrupt – usually 100 for x86

67 And this information: http://lwn.net/Articles/105136/

68 And local policies are not often implemented (VMS is one exception) – they are hard to tune to make optimal use of system resources and if made auto-tuning tend to mimic global policy instead

69 Linux source code is developed in several independent trees maintained by influential developers to test different (and conflicting) kernel features. The most prominent ones are the -mm tree maintained by Andrew Morton, the -rmap tree maintained by Rik van Riel and the mainline tree maintain by Linus Torvalds

the swap prefetching; in addition, a limited number of pages paged out most recently is remembered – it is assumed that these are the pages that will be needed the soonest. The thread wakes up periodically to perform the prefetch but goes back to sleep immediately if it detects a high memory activity – the criteria include free memory amount, number of dirty pages, disk writeback in progress or the swap cache size. If the thread concludes it is safe to proceed with the prefetch, read pages are placed to the end of `inactive_list` and their copies kept on the backing store – in this way, they will be the first to be paged out, and cheaply too, in case memory becomes scarce soon.

As a demonstration of the positive influence of swap prefetching (although this would be obviously best demonstrated with an interactive application and user's experience), we will write a short program that allocates a block of memory and then forks off a child which does something memory hungry and causes the parent's data to be paged out. This simulates an inactive application having its working set paged out by unrelated activity in the system. The parent just waits for the child to complete and goes to sleep to give the swap prefetching algorithm a chance. Finally, the parent wakes up and measures the latency of accessing every page of the original allocated block. We will use Con Kolivas' swap prefetching patch[70] to the 2.6.18 kernel revision. With prefetching disabled[71], the kernel would have to bring most of the pages from the swap area; while with prefetching enabled it would, hopefully, find most of the desired pages already in main memory.

Kernel	**Average Time**
2.6.18 [swap prefetching disabled]	9.7s
2.6.18 [swap prefetching enabled]	4.1s

This particular case had the parent allocate 20% of all available memory before forking off a child, which in turn allocated 150% of available memory before exiting. The prefetching feature decreased the time required to re-access the original memory block to less than a half. The parent slept for one minute before performing the re-access, this was enough for the swap prefetch to bring in all its remembered pages (among them, approximately 75% of our buffer's paged out portion) – other applications, with different fractions of their memory in the prefetcher's remembered pool will benefit accordingly.

70 http://ck.kolivas.org/patches/swap-prefetch/2.6.18-rc2-swap_prefetch-33.patch
71 Through the `sysctl () swap_prefetch` variable

Another issue with the memory manager's design is the increasing cost, both in time and the amount of memory used, of accessing page tables [Szmajda03]. 64bit address space machines require slow and expensive page table structures; moreover, modern CPUs became much faster than main memory making the effect of TLB misses much worse than in the past. This is even aggravated by the rising memory capacity because TLBs can now cache a much lesser portion of available memory. Making TLBs larger yields diminishing returns as TLBs need to be invalidated fairly often; besides, sophisticated CAM memory is required for their construction, which is very hard to be made large, fast and cool. Other factors contribute to the issue; for example, while memory sharing can be recognised and optimised by both main memory and CPU caches (the physically indexed ones, of course), each of the sharing processes requires a separate TLB entry for a shared page[72].

Larger pages are one possibility of improving this situation – by keeping page tables smaller and TLB hit rates higher. However, larger pages cause fragmentation problems and decreased I/O bandwidth, so it is desirable to be able to use different page sizes – each for a different purpose. Some architectures may provide hardware support for pages of different sizes[73] or contiguous pages can be clustered together and treated as one superpage in software.

This clustering is inefficient though, and it is much more desirable to have hardware do the work, which can be considerably simplified by a suitable page table structure. For example, x86 can regard the lowest page directory entry as a page table entry mapping 4 MB of memory. This is how the kernel creates its page tables as mentioned earlier.

But this structure is rigid, x86 with two levels of page tables cannot support more than two different page sizes this easily. The variable radix page table was designed to address this shortcoming. Outwardly, it is a forward-mapped page table, which however allows for a virtual address to be split into a different number of fields of varying lengths (contrary to the fixed 10 bits for the directories, 12 bits for the page table used in a traditional “fixed radix page table” on x86). Consequently, different depths of page tables can be used for different parts of an address space, easily allowing the use of superpages when required.

Regarding the page table structure, there has been an effort to push the currently used page tables in linux to the architecture dependant layer of x86 and provide an architecture independent

72 Though there are advanced TLB designs with tags that do not identify an address space but a protection domain shared by many distinct address spaces

73 x86 provides only two – 4 kB and 4 MB (2 MB with PAE enabled), but machines with much better support exist – IA64 provides a total of 11 different page sizes ranging from 4 kB to 4 GB

interface allowing for easy reimplementation of different page tables for each architecture. This has been specifically proposed with variable radix page tables in mind (for the IA64 machines). More information on the project can be found on the Gelato web page[74].

74 http://www.gelato.unsw.edu.au/IA64wiki/PageTableInterface

7 Conclusion

The work presents a condensed view on memory management in general and its linux implementation in particular. Many details of description have been omitted for the sake of brevity, most of the topics included in Section 6 were merely hinted on as each would deserve a work of its own. Nevertheless, we believe the work offers a coherent account of the topic and may serve as an introduction to the domain of memory management.

With this said, the work cannot honestly pretend to represent original contribution. All algorithms, data structures and approaches discussed are well studied and proven solutions. The behaviour of the linux kernel under many imaginable conditions is well known and tested, all the unofficial patches mentioned have both rationale and test results backing their claims and well understood and described limitations on the ground of which they were denied merging into the mainline kernel. The work provides merely a summary of these scattered facts.

Possible improvements and ideas for future revisions include a more thorough description of the page eviction algorithm implemented in linux and conducting more detailed benchmarks. It would probably be interesting to perform some regression tests on the linux kernel, explore the impact of different hashing functions used in kernel space or determine the performance of the buddy system allocator variations.

Of works similar in topic and approach, we would like to mention Mel Gorman's description and detailed, line by line, code commentary of the linux memory manager [Gorman04]. Interested reader can consult the book for further details.

8 Abbreviations

APIC	Advanced programmable interrupt controller
ARC	Adaptive replacement cache
AVL	Adelson-Velsky, Landis
BSD	Berkeley Software Distribution
CAM	Content-addressable memory
CPU	Central processing unit
DDR	Dual data rate
DMA	Direct memory access
FIFO	First-in, first-out
GNU	GNU's not UNIX
GPL	GNU Public Licence
IEEE	Institute of Electrical and Electronics Engineers
I/O	Input/output
IPC	Inter-process communication
IRR	Inter-reference recency
LFU	Least frequently used
LIRS	Low inter-reference recency set
LRU	Least recently used
MIPS	Multiprocessor without interlocked pipeline stages
MMU	Memory management unit
NRU	Not recently used
NUMA	Non-uniform memory access
PAE	Physical address extension
PGD	Page global directory
PMD	Page middle directory
PTE	Page table entry
PUD	Page upper directory
POSIX	Portable Operating System Interface
RAM	Random access memory
RISC	Reduced instruction-set computer
ROM	Read-only memory
SMP	Symmetrical multiprocessing
SUS	Single UNIX Specification
SVR2	System V Release 2
TLB	Translation look-aside buffer
UMA	Uniform memory access
VFS	Virtual file system

9 References

[Bach86] Maurice J. Bach, “The Design of the UNIX Operating System”, Prentice Hall – 1986

[Bonwick94] Jeff Bonwick, “The Slab Allocator: An Object-Caching Kernel Memory Allocator”, Proceedings of the USENIX Summer 1994 Technical Conference, pp. 87-98 - 1994

[Carr81] Richard W. Carr and John L. Hennessy, “WSClock – A Simple and Effective Algorithm for Virtual memory Management”, Proceedings of the 8th ACM symposium on Operating Systems Principles, pp. 87-95, Pacific Grove, California - 1981

[Denning68] Peter J. Denning, “The Working Set Model for Program Behavior”, Communications of the ACM, Volume 11, Issue 5, pp. 323-333 – May 1968

[Fotheringham61] John Fotheringham, “Dynamic Storage Allocation in the Altas Computer, Including an Automatic Use of Backing Store”, ACM Communications 4, 10, pp. 435-436 – October 1961

[Gorman04] Mel Gorman, “Understanding the Linux Virtual Memory Manager”, Prentice Hall - 2004

[Huck93] Jerry Huck and Jim Hays, “Architectural Support for Translation Table Management in Large Address Space Machines”, Proceedings of the 20th Annual International Symposium on Computer Architecture, pp. 39-50, San Diego, California – 1993

[Jiang02] Song Jiang and Xiaodong Zhang, “LIRS: An Efficient Low Intere-reference Recency Set Replacement To Improve Buffer Cache Performance”, Proceedings of the 2002 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Marina Del Rey – California – June, 2002

[Jiang05] Song Jiang, Feng Chen and Xiaodong Zhang, “CLOCK-Pro: An Effective Improvement of the CLOCK Replacement”, Proceedings of 2005 USENIX Annual Technical Conference, Anaheim, CA - April, 2005

[Jiang05a] Song Jiang and Xiaodong Zhang, “Token-ordered LRU: An Effective Page Replacement Policy And Its Implementation in Linux Systems”, Performance Evaluation, Vol. 60, Issue 1-4, pp. 5 – 29, 2005

[Knowlton65] Kenneth C. Knowlton, “A Fast Storage Allocator”, ACM Communications 8, 10, pp. 623 – 624 – October 1965

[McCalpin95] John D. McCalpin, “Memory Bandwidth and Machine Balance in Current High Performance Computers”, IEEE Technical Committee on Computer Architecture – Septermber 1995

[McVoy96] Larry W. McVoy and Carl Staelin, “lmbench: Portable Tools for Performance Analysis”, Proceedings of the USENIX 1996 Annual Technical Conference - San Diego, California, January 1996

[Megiddo03] Nimrod Megiddo and Dharmendra S. Modha, “ARC: A Self-Tuning, Low Overhead Replacement Cache”, Proceedings of the USENIX File & Storage Technologies Conference (FAST) – San Francisco, CA – March, 2003

[Nagle93] David Nagle, Richard Uhlig, Tim Stanley, Stuart Sechret, Trevor Mudge and Richard Brown, “Design Tradeoffs for Software-Managed TLBs”, Proceedings of the 20th Annual International Symposium on Computer Architecture, pp. 27-38 – May 1993

[O'Neil93] Elizabeth O'Neil, Gerhard Weikum and Patrick O'Neil, “The LRU-K Page-Replacement Algorithm for Database Disk Buffering”, Proceedings of the ACM SIGMOD Conference, pp. 296-306, Washington, D.C. - May, 1993

[Saavedra95] Rafael H. Saavedra and Alan Jay Smith, “Measuring Cache and TLB Performance and Their Effect on Benchmark Runtimes”, IEEE Transactions on Computers, 44 (10), pp. 1223-1235 – October 1995

[Sears00] Chris B. Sears, “The Elements of Cache Programming Style”, Proceedings of the 4th Annual Showcase and Conference, pp. 283-298, Berkeley, CA – October 2000

[Szmajda03] Cristan Szmajda and Gernot Heiser, “Variable Radix Page Table: A Page Table for Modern Architectures”, Asia-Pacific conference on advances in computer systems architecture 2003, pp. 290-304 – September, 2003

[Tanenbaum01] Andrew S. Tanenbaum, “Modern Operating Systems”, second edition, Prentice Hall – February 2001

[Vahalia96] Uresh Vahalia, “Unix Internals: The New Frontiers”, Prentice Hall - 1996

10 Appendix: CD Contents

Benchmarks source code:

memory_hog.c

mmap.c

sequential_scan.c

swap_prefetch.c

thrashing.c

Makefile

Complete lmbench results in accordance with the program's licence agreement:

lmbench_results.dat

Source code of third party benchmarks:

lmbench-3.0-a7.tar.bz2

Kernel patches tested:

swap_prefetch33-2.6.18-rc2.patch

clock_pro1-2.6.18-rc5.patch