e-con Systems

17, 54th Street,
Ashok Nagar
Chennai-600083
www.e-consystems.com

Porting Linux Memory Management Unit onto a new Processor

e-conIN0401WP

Whitepaper

Version 1.1

Customer/Partner
Internal

Revision History

Rev No	Date	Major Changes	Author
Initial Draft	May 6, 2009	Initial Draft Version	Maharajan

Author

Maharajan Veerabahu
Project Manager,
e-con Systems India Pvt. Ltd

PORTING GUIDE FOR LINUX Memory Management Unit (MMU)

Preface

Linux is being ported on to much new architecture. Its open source and extensive support has made it the best choice as far as porting is concerned. One of the most interesting and tricky of Linux modules is the Linux MMU and porting it is a good challenge for programmers. The basic idea behind this document is to provide a detailed analysis of the Linux MMU and to explain the necessary steps to be taken when porting it onto a new platform.

Audience

This document will be useful for newbie's in the porting and also serve as a good reference to others who may look for the exact functions and files to be modified for porting the MMU of Linux. The document comprehensively covers the basics and also ponders on the functions and files that need modification in the Linux source code when porting.

Documentation Style

The author has followed a narrative style of documentation with comments at places for the liveliness of the document. Figures are added where ever necessary. The document is organized as three main parts. To maintain the flow, it is advisable to read them in the same order as in the document.

TODO

Linux MMU is an ocean and with this document we are picking only the small pebbles along the shore. This document refers to Linux 2.4. In course of time this document will be updated with more examples and for different processor architectures. A document covering the MMU in 2.6 is also in the cards.

Feedback

Any kind of feedback which will help us improve or correct will be of great help to the us and Linux community. Feedbacks are welcome.

INTRODUCTION

To port the MMU code of Linux onto a new processor environment will require the following pre-requisites. Generally to port anything we should know the answer to the following basic questions what, on what, how to port and knowing that the porting job is considered half complete.

1. A complete understanding of the hardware MMU of the processor and the platform. (On what?)

    The level of paging.
    The page fault exception generation and error codes corresponding to it.
    The root table pointer or pointers.
    The memory map.
    Flags associated with the page tables.
    Cache and TLB architecture and type.

2. A clear understanding of the Linux MMU handling (What?)

    Linux paging basics
    The User & Kernel pages setup
    Process Address spaces, memory regions. Page fault Handling.
    Cache and TLB Handling

3. The coding part that has to be changed in LINUX source (How?)

This document helps you in resolving the answers to the basic questions. Although the coding part to be changed (How?) the one of interest we deal with the Linux MMU handling elaborately as that will give the reader a very strong foundation on the porting job and thereby help him resolve critical bugs (if any). As we are not into writing a guide for a particular architecture the hardware portion has to be filled in by the reader with the processor of his choice by reading the respective hardware manuals. Anyway general hardware concepts are outlined and a short note on a sample processor is included.

UNDERSTANDING THE HARDWARE

General

In almost all the processors running an operating system, have a Memory Management Unit. In the hardware under consideration, namely the 32 Bit RISC processor we have the paging mechanism the heart of the Linux MMU.

The processor has a demand paged MMU.

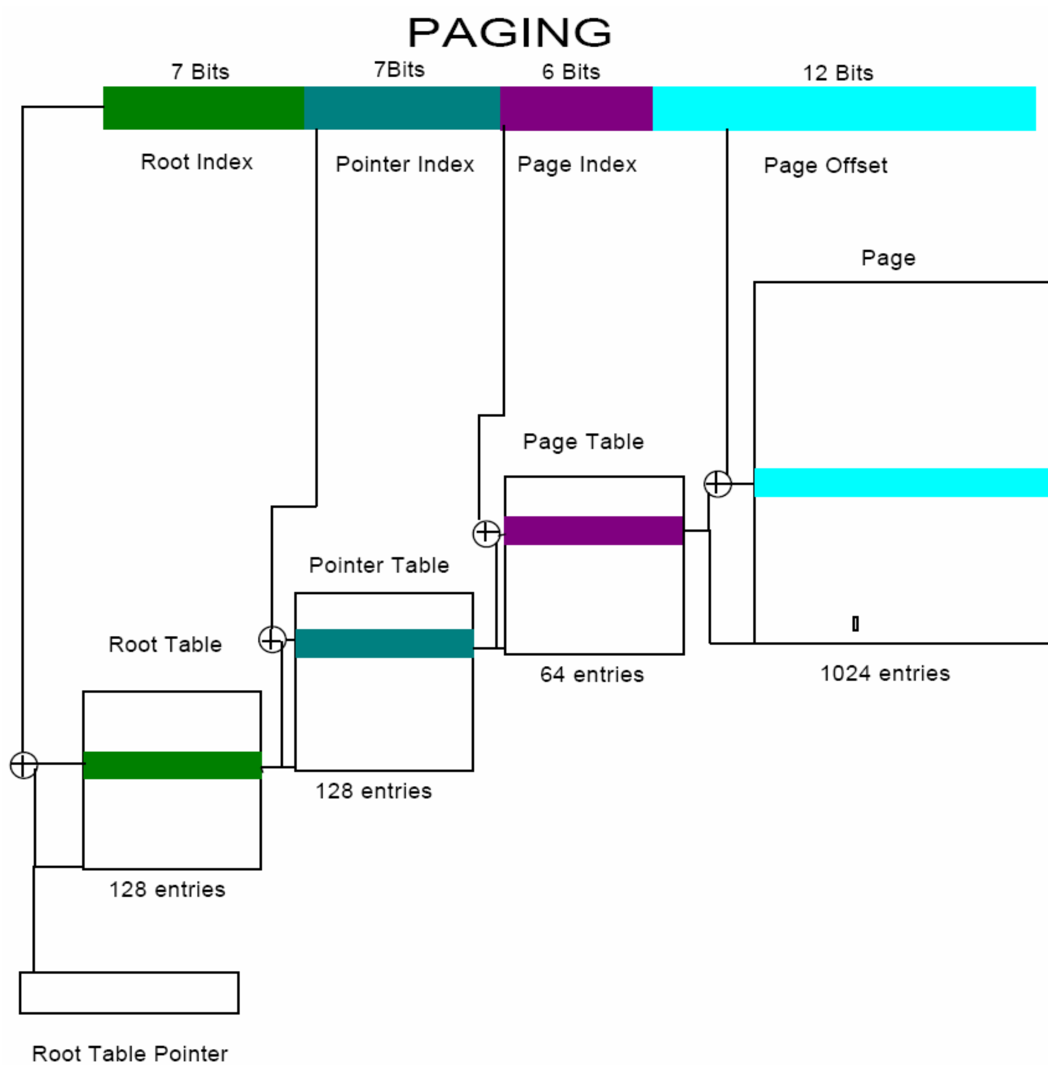


FIG 1

PAGING:

The paging concept is the heart of virtual memory system. In paging all the addresses are logical addresses but not physical. The paging unit with the help of tables translates the logical addresses into physical addresses. For ease of operation the smallest unit in this paging will be a page which is generally 4KB (some cases 4MB). A page is a set of continuous linear addresses and the data contained in them. A page frame will be a set of continuous physical addresses in the RAM into which a page can fit. The whole RAM is assumed to be divided into a number of pages frames from the MMU's point of view.

In the processor we are discussing the MMU translation occurs in 3 stages based on 3 tables namely the root table, pointer table and the page table. (FIG 1). The next address after the MMU is switched ON (by setting a bit in the appropriate register) will go for a translation.

The 32 Bit linear address is divided into 7-7-6-12. There are registers called User Root Pointer (URP) and Supervisor Root Pointer (SRP) which point to the root tables of the user and supervisor respectively. The first 7 bits in the 32 bit split up shown above is the index into the 128 entry root table, where the base of the pointer table is found. The next 7 bits is the index into the 128-entry pointer table, where the base of the page table is found. The next 6 bits is the index into the 64 entry page table, where the base of the 4K page is found and the last 12 bits is the index into this page, which shows the physical address and where the 32 bit data is found. The figure provides the idea behind the paging methodology.

Specific

We have to understand the hardware platform to which we are to port Linux, completely. Throughout this document we will be using a desktop based on a imaginary 32-bit RISC processor called MODEL. The MMU part of this is based on the Motorola core. A short note on the MMU hardware is given here.

The MODEL processor has a demand-paged virtual memory system(Pages are given to programs only when they demand by accessing the linear address) .

The following are the salient features of MODEL

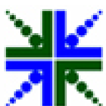
- Independent Instruction MMU and Data MMU
- 32- Bit logical address translation to 32 - Bit physical address.
- 64-Entry-4 way- set associative Translation Look Aside buffers.
- 4KB page size.
- Two - 16KB on-chip cache (copy back/write-through modes supported)
- Two separate root pointers (URP and SRP) user root pointer and superuser root pointer.
- Three level Page tables (Root table, Pointer Table, Page table) (Is exactly the same to the paging described earlier)
- Page descriptor format

This is necessary to understand the flag setting in the PTE BITS Meaning

31 -12	Physical address
12, 11	Reserved for user
10	Global (Whether Global, Used for flushing operations)
7	Superuser /User
6, 5	Cache mode (00 - write-through, 10 - Copy-back, 11 - inhibited)
4	Modified
3	Used
2	Write-protected
8,9,0,1	Not -Used.

The desktop that we use is similar to the normal desktop but has the MODEL processor running at its heart. The amount of RAM would be 512 MB.

This a very short description of the environment and the processor. This is given here just to give a feel and also to mark the thing that has to be noted in any new processor environment. The best suggestion while porting the MMU part of Linux would be to make a small table containing all the above mentioned facts about the new processor of your choice. Additionally you can include the other registers related with MMU, the special MMU instructions, warnings if any, etc. Apart from the manual this table will help a lot for your quick references and while troubleshooting MMU bugs in the kernel you are trying to bring up.



The Root pointer confusion

The virtual address space that Linux sees is 4GB. This is divided into 3GB for User and 1GB for Kernel. This is because the x86 architecture has only one root table pointer namely cr3 register (contrast to the MODEL which has URP and SRP). In x86 whenever a translation has to be done the root table base is taken from the same register irrespective of whether the translation is for user or kernel, but in case of MODEL the root table base is taken from URP in case of user and SRP in case of kernel. Since the the Intel model was forced to have one common space for user and kernel, the Linux mmu was designed as a 3GB / 1GB model as the user space required is always more, especially in case of server applications. The 4th GB of the address space belongs to the kernel. All user processes can address memory 0 - 3GB only. But in case of MODEL we can very well have a 4GB / 4GB as both root pointers are different completely, thereby solving many problems arising due to the kernel 1GB restriction. But in this document all the problems arising with the 3GB/1GB model will be addressed so that a new processor of that variety can be handled with ease(just in case).Please note that these problems mentioned now and then in the document do not pertain to MODEL.

The hardware and terminology

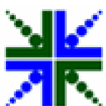
The MODEL processor has been chosen and set contrast to Intel x86 purposely to add a different vision to the MMU from Intel. A mention about the X86 will occur here and there in the document so that all concepts are covered (like the root pointer mentioned above).

The words Root Table, Page Global Directory, PGD all are one and the same

The words Pointer Table, Page middle Directory, PMD all are one and the same

The words Page Table, Page Directory, PT all are one and the same.

These will be used mixedly in this document.



THE LINUX MMU HANDLING

Having analyzed the hardware, the most important part of Linux memory management unit is handled in this chapter.

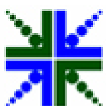
Many processors have a peripheral unit within called the Memory Management Unit which helps in the maintaining the memory. Linux makes use of this MMU inside the processor to do the jugglery of memory management. The memory management hardware just provides means to use the virtual addressing, paging etc. Its up-to the operating systems to set up page tables and appropriate access rights and make complete use of the hardware. Linux does this is a rather astounding way. The aim of this document is to provide a clear insight into the Linux MMU and the source code related to the MMU facilitate the porting of Linux onto new environments. Linux has two modes of operation the Super user or Kernel mode of operations and the User mode of operations. The memory resource can also be seen as kernel and user memory (pages). Firstly, we will see the kernel page handling. The Linux while boot-up is in complete kernel mode and uses the kernel page tables.

The KERNEL Address space

The kernel pages are different from the user pages only in access rights.

Creation of the kernel page tables is more or less the first step in the Linux boot process. At the start only a few tables are setup (generally for 8MB). This is to facilitate the kernel install itself in the RAM. Once the pages are set the paging is switched ON and the addresses that follow undergo translation based on the tables set.

Caution: The next address encountered after the instruction that switches ON the paging, may undergo translation in most architectures. The kernel virtual address space ranges from 0xC0000000 - 0xFFFFFFFF (4th GB) and the mappings done were for that range (0xC0000000 mapped to 0x00000000). But the Linux code runs from 0x100000 physical address generally. So for eg. if the paging was switched ON by the instruction at 0x100040, the next address 0x100044 may undergo translation only to find no tables resulting in a fault causing a fatal kernel halt at the



very start of the boot process. To avoid this generally 2 methods are possible.

** The same physical address, 8MB (0x00000000 to ox7FFFFFFF) are mapped to two virtual address spaces starting at 0x00000000 and 0xC00000000.*

** Somehow tell the MMU hardware not to translate a particular range of addresses (Hardware support is essential for this)*

Once the 8MB mappings are done the Kernel Master Page Global directory will have only one entry as one entry in the global (root) directory can map 32MB ($1 * 2^7 * 2^6 * 2^{12}$) in Abacus. All other entries in the global directory are made zeroes.

After this initial setup, the kernel code is safe from memory faults and can continue to run. Once this basic requirement is fulfilled, the building up of kernel page tables for the entire available RAM is carried out. This process has a few issues associated with it.

1. The RAM is less than 896 MB.
2. The RAM is greater than 896 MB but less than 4GB.
3. The RAM is greater than 4GB.

The size of the RAM is an important criteria, as we are speaking of mapping the entire physical RAM into kernel pages. The kernel virtual address space is limited to 1GB and a RAM more than 1GB cannot be mapped easily as we map RAM less than a GB. For RAM greater than 4GB, we lack address space rather address lines (only 32).
Now the situations in detail.

RAM < 896 MB

This is normal and the kernel page tables are created normally for the entire RAM. The mapping starts from 0xC0800000 (mapped to physical 0x00800000) and proceeds till the last physical RAM address. The Kernel Page Global Directory now has many entries apart from the one we created at the start. All the pages have the Present, Access, Superuser, dirty, Read/Write flags set. This makes sure that the user cannot access the kernel address space. The 896 MB limitation is to use the rest 128MB of the kernel address space for fixed mapped pages and Non-contiguous memory mappings.

896 MB > RAM < 4 GB

This situation is a cause of concern for the x86 processors as they have only one common pointer for user and kernel and hence the 3GB /1GB restriction. In these situations, the high address frames are mapped using one of the following techniques.

1. Permanent kernel mappings
2. Temporary Kernel Mappings
3. Non - Contiguous Kernel Mappings.

Permanent Kernel Mappings

The permanent kernel mappings as the name implies, establishes a long-lasting mappings of high memory page frames into the kernel address space. The Linux kernel makes use of a dedicated page table (pkmap_page_table) for this purpose. There are as usual 1024 entries in this pkmap_page_table. The linear addresses where this high page frames are mapped starts from 0xFE000000.

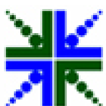
There are counters associated with each entry in the pkmap_page_table. The counter values depict certain states of that page table entry.

Counter = 0: The page table entry does not map any high memory page frames. It can be used for mapping high-mem pages.

Counter = 1: The page table entry does not map any high memory page frame but is not usable as the TLB is not flushed.

Counter = n (>1): The page table entry maps a kernel high-mem page frame which is used by (n-1) kernel components.

While mapping the high-mem page frames, the pkmap_page_table is taken and the counters corresponding to the page table entries are checked for 0 so that a new mapping can be made. The page table entries with counter value 1 are found and their TLBs are flushed so that they can have a counter value of 0 and can be used in the next mapping. In case no free page table entries are found the process sleeps until it gets one free entry.



Due to this the current process may be blocked for a long time and thats why permanent kernel mappings are NOT solutions inside interrupt handlers and deferrable paths.

The kmap() and kunmap() functions are responsible for mapping and unmapping the page frames and these call kmap_high and kumap_high if they find out that the mappings are for high-mem area.

Temporary Kernel Mappings

The temporary kernel mappings are more easier than the permanent kernel mappings and they are non-blocking and can be used inside interrupt handlers. Any page frame in high-mem can be mapped through a small window in the kernel address space. The numbers of such windows are very small. Generally, each CPU in a system will have a set of 5 windows of linear addresses. As the same window should not be used by more than one kernel path, unique names are given to the 5 windows corresponding to the kernel path that will use them. Actually these 5 windows are stored in the form of indices to fixed-mapped-linear addresses.

A fixed mapped linear address is a constant linear address that can map any arbitrary physical address. They are different from normal linear addresses (the ones that map the first 896 MB) in one way that they do not show any linearity in the mapping (Linear address A always maps physical address A- PageOffset in the first 896 MB mapping). Fixed Linear address can map any physical address. The main point behind the fixed=mapped linear addresses is that while compile time we can have a constant address and allocate the physical addresses only at boot time.

The mapping function finds the correct fixed mapped virtual address using certain parameters and sets the page table entry corresponding to the fixed map linear address with the required high-mem physical address. The functions used for temporary kernel mappings are kmap_atomic( ) and kunmap_atomic( ).

The temporary kernel mappings are very rare and are used only when high-mem needs to be mapped inside interrupt handlers and fixed mapped linear addresses are equally rare in uni Processor systems.

Non - Contiguous Memory Mappings

It's a good idea to map contiguous page frames to contiguous linear addresses. But if memory requests are very infrequent non-contiguous mapping will be of use. The idea is mapping non contiguous physical memory into contiguous linear addresses. The linear addresses offered in the kernel linear address space for Non contiguous memory areas is after the first 896 MB mappings of RAM and before the fixed kernel mappings and Permanent kernel mappings.

The figure will give a clear picture about the exact location of the linear addresses used for non-contiguous memory mappings.

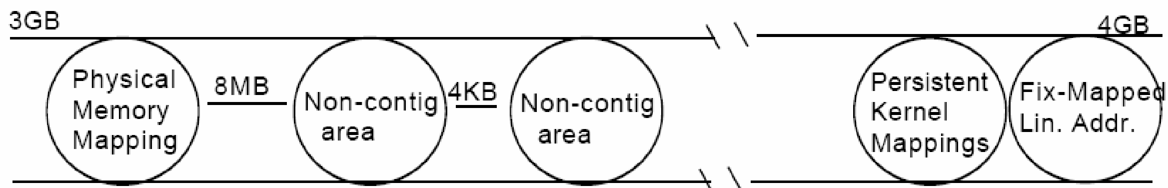


FIG 2

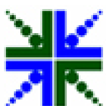
The mappings are updated in the master kernel page global directory. The mappings are done by vmalloc( ) and freeing is done by vfree( ).

RAM > 4 GB

In this case the hardware should support the Physical Address Extension (PAE) and the kernel code should be compiled with PAE enabled. All processors that support the PAE also supports the 2MB page size and Linux makes use of these hardware privileges to provided the kernel mappings.

As our processor does not support PAE we will not discuss this in detail here.

The important fact to be noted is that the kernel's demand for dynamic memory is never deferred and given to it at any cost as the kernel trusts itself.



USER Address space

The User processes have user page tables and the space they can access is 3 GB as mentioned earlier. Initially there are no page tables set in case of user, contrast to that of the kernel where as we saw in the earlier part the page tables are set at the startup itself. The user pages and their page tables come into picture after a comparatively long period in the boot process. The thought about the user pages arise only after the first user process runs ands faults. Before this the kernel page tables are set and those page frames used by the kernel code and data are marked reserved. Those reserved pages are no longer considered for the dynamic memory allocation or for swapping.

The USER Process Address Space

A process can request dynamic memory at many stages but allocation of this costly and protected resource is not straightforward as it was for the kernel, the ultimate master of events. The simple reason is that the process is not going to make immediate use of all the memory it requests (Its always greedy and lavish??!!) and so the allocation of real memory can be deferred as much as possible. Also programs may be buggy, accessing forbidden areas of memory and may cause serious problems to the kernel and the system as a whole, a thing that is never expected from a user process. So such violations can be checked with the deferred approach. As a result of this deferring whenever a user process requests for dynamic memory its not given page frames from the RAM(remember for the kernel dynamic memory request page frames are given freely at will), instead, its given the right to access a new range of linear addresses. This interval of linear addresses is called a memory region. The kernel can dynamically modify a process address space by adding or removing memory regions. Its an utmost necessity for the kernel to know the memory regions owned by the current process. When a page fault occurs the kernel can decide judiciously, whether to allocate a page frame (i.e. the faulting linear address is in the memory region of the process but no page frame is allocated -- very much legal) or terminate the process (the faulting address does not belong to the memory region of the process -- Illegal). So all the information related to the process address space is stored in a structure called the memory descriptor which is referenced by the mm field of the

process descriptor. The usage of memory descriptor takes an interesting turn in-case of kernel threads. Although they resemble processes, kernel threads never have the necessity to use the memory regions as they are superusers and get page frames directly. As a result most of the fields in the memory descriptor are meaningless for kernel threads. Also, the kernel thread is going to address only the kernel address space (addresses > 0xC0000000 in Intel). That part of the address space is going to be the same for any process and so it really does not matter what set of page tables kernel threads use.So they use the page tables of a standard process. Exclusively for this reason there are 2 entries in the process descriptor namely mm and active_mm.

mm --> Memory descriptor owned by the process.

active_mm --> Memory descriptor used by the process when it is in execution.

Well both of these are one and the same for normal processes. But in the case of kernel threads the mm field is null as reasoned above and the active_mm points to the previous process (Well any table is okay for the kernel).

Having seen about the memory descriptors, next we'll see the memory regions in detail. The memory region descriptor generally denotes a linear address interval. The start address and the size of the memory regions are always multiples of 4K. No two memory regions can overlap in a process. Care is taken to extend a memory region if a new memory region is allocated just next to the existing one. The kernel tries to resize the region accordingly when memory regions are removed.

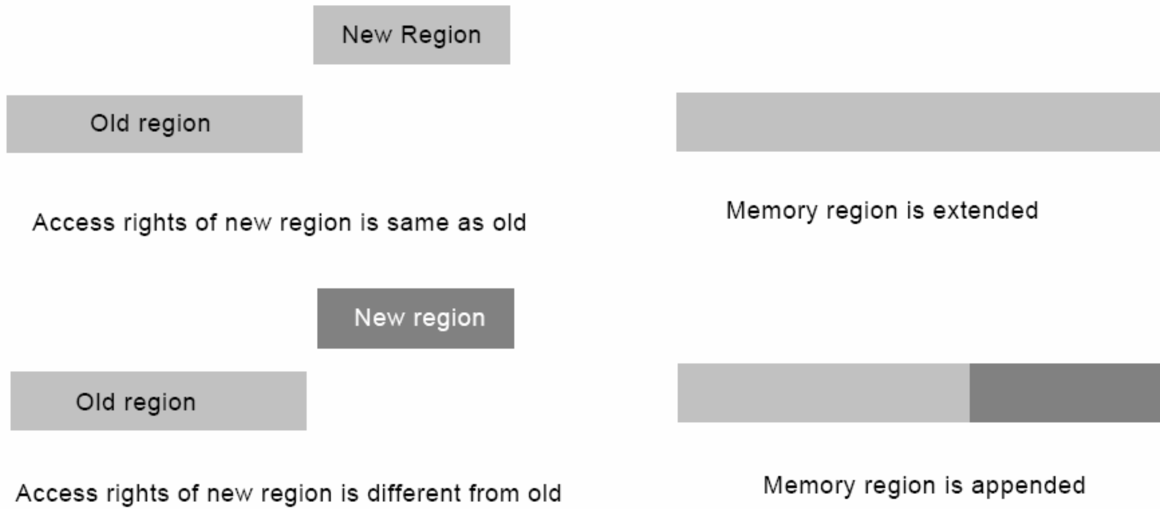


FIg (a) above : MEMORY REGION ADDITION

Fib (b) below : MEMORY REGION DELETION

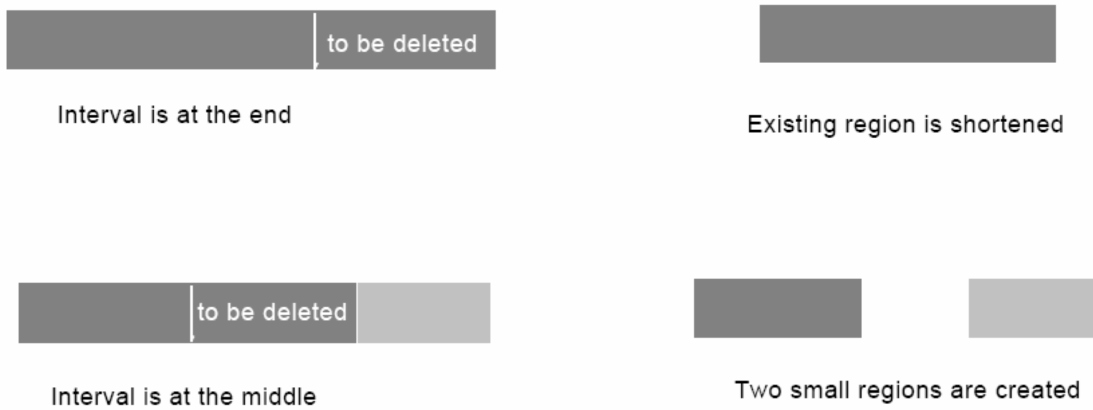


FIG 3

Getting hold of memory regions

Allocation of memory regions is a bit complex process as the kernel has to look for so many criteria and act accordingly. For instance it has to check whether the requested address is valid, is that of a file, if so what the permissions are is it the first one for that process, is the region Merge-able is it for shared memory and so on. A detailed commentary of the events involved in this allocation of memory regions is given below.

The allocation of memory regions starts with the checking the validity of the request for the memory region. The request may be termed invalid at the first place if the linear address length is zero or more than TASK_SIZE (3GB), if the process has mapped too many memory regions (No more memory for you, buddy!!), the request is for a locked in RAM area and the process exceeds the limit of locked RAM pages (You can't take the whole of RAM!!)

Locked pages in RAM are resident in RAM always and are not swapped and the page table entries are always present(created at memory region allocation time itself)

It also checks the offset of the file, if the new memory region mapped to a file in disk. A check is made on the address field if the request insists that it needs the initial address to be the one it wants and the address is returned if the checking proves okay. In the case that the request is not insisting on the initial address returned, a search is made in all the virtual memory regions owned by the process using the memory descriptor to find a suitable initial address which does not overlap any of the memory regions already owned by the process.

Now having found the starting linear address of the memory region, the next thing of importance will be the access rights of the memory region. The memory access flags are calculated carefully and are set appropriately to the memory region. Although we have decided the starting linear address there may still be a problem with the ending address. In-case the request was for a size that makes

Starting address + size > PAGE_OFFSET (0xC0000000) [Oops!! i cant give you this region. sorry!!!}

So a check is made and proceeds only if the result is less the PAGE_OFFSET.

Generally when a memory region request rises, care is not taken to assure whether there are enough free page frames in the RAM. And this is re-iterated by a flag sent by the requesting function. But in some rare cases the requesting function may want to check for enough free page frames. In that case the free page frames are checked and if there are not enough the request fails.

Finally all the checks are over and we have a valid memory region. The step next will be to try and merge this region with an existing memory region. (That will make a lot of jobs easy). This can be done only if the region is not for shared access. not part of a file, has the same access rights of the region its going to get merged.

If the memory region requested is to map a file in the disk, the permission of the files are also checked.

Once all these are done the memory region descriptor is got and all the fields are initialized based on the steps carried so far.

PAGE - FAULT HANDLER

Kernel Access to Kernel Space

Non-Contig Memory Area address

In interrupt or kernel thread

Kernel Page Table Entry Fixup

Address in memory region

Address can belong to USER stack

Write Access

In USER mode

Page is Present

Region is Writable

Region is readable or executable

Address is wrong sys.call parameter

Copy On Write (COW)

Demand Paging

Send SIGSEGV

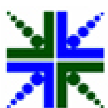
Kill process & Kernel " Oops "

Fixup - code

YES

NO

Figure 4



The PAGE FAULT Handling

The page fault handler's main role is to handle the fault generated when a process tries to access a legitimate page which belongs to the process address space but has not been allocated yet and meanwhile catch the odd faults caused by programming errors. The page fault can be better understood with the help of a flow diagram given below

The first thing to be checked is the linear address where the fault occurred. Its checked whether the address is from the user space or kernel space. In case of kernel faults the only reason behind a valid fault may be because of the non-contiguous mapping which reflects on the master kernel page global directory but not on all process page global directory. So a check is made on the Master Kernel Page Global Directory. If the entry is available it is copied to the page global directory of the process. Else its a no-context fault in kernel space and the kernel goes to a halt. The same method is repeated for the master page middle directory and page tables.

It is also checked whether the fault has occurred inside a interrupt or kernel threads. As we have seen earlier interrupts and kernel threads always use addresses greater than the PAGE_OFFSET (kernel addresses) and their memory region descriptor is of no use. So we have no context and the process is killed for a wrong fault of it. After passing through this the handler checks whether the linear address is a part of any memory region obtained by the process. It does this by going through the list of memory regions pointed by the memory descriptor. If it finds a valid linear address it is termed a good-area fault and that starts the real work of the page fault handler. In case it fails to find the linear address in the memory regions it still cannot determine it as a no-context fault because it has to do one last check whether its a stack extension fault. This is because that the memory regions include the stack but cannot delimit the stack. In case of a stack extension fault the memory region of the stack is extended (the start field is reduced as it is always down-growing stack) and then its termed good area and fault is handled.

Having found a good area where the fault has occurred the handler follows a series of steps. As a first step after identifying a good area the handler checks the 2 bit error code

(for i386, 3 bit) for the reason behind the fault. If the reason is a write fault the memory region flags are checked to find whether a write access is allowed, if not it's termed a bad area access and the process is killed. If the reason is a read/execute fault it's checked whether the page is in RAM already. If it exists then the fault and is due to access violation and again the process is killed. If it does not exist then a check is made to see whether the region allows read/execute and confirms a good area fault. Finally if it is decided that its a good area fault then the steps to provide a page frame are taken. First of all it checks whether a page middle directory and a page table exists to map the current memory region. If it does not exist they are created (Of course, the tables are the initial requirement). Once that is done the allocation of page frames is carried out. The allocation of page frames can take either of the two courses.

1. If the page is not in memory (i.e. its not in any page frame and is new) then the kernel looks for a free page frame and initializes accordingly and returns.
2. If the page is in memory (i.e. if the page frame already exists but has only read permissions) the contents are copied to another new free page frame by the kernel.

The first one is called ***demand paging*** and the latter is the ***Copy On write (COW)***.

Demand Paging:

Demand paging has been the best of all dynamic memory allocation techniques. It defers the allocation of a page frame to the longest time possible. Only when the process addresses a memory region and finds it not to be in memory and raises a page fault, the demand paging mechanism will allocate a page frame (After-all ! it has to allocate sometime ..). This has increased the is lesser no. of ideal page frames and makes the user feel a better memory performance even with lesser amount of RAM (thats what i call resource management).

Now will analyze the nuances in demand paging.

An addressed page may not be in RAM for 2 reasons basically,

1. It was never addressed before by the process.

2. It was addressed but swapped to the disk. [Handled later in this tutorial]

In case 1, where the page was never addressed before, there are further two ways of handling,

Type 1: The 'never addressed before' page maps a file in the disk. In this case a pointer in the memory region descriptor points to a function that loads the file which has to be mapped to the page. This function based on the file-system generally checks the page cache for the page and if it fails does the necessary steps (based on file system again) to read the page from the disk and fill it in the free page frame alloted by the kernel and dutifully updates the page table entry in the page table with the address of the page frame. (And there's your page in memory!!)

Type 2: The 'never addressed page' does not map a file to memory (anonymous page). In this case the pointer in the memory region pointing to a function that loads the file is NULL. Now handling of this anonymous page is different for read and write. While handling a write access a new free page frame is got by the kernel, filled with zeroes and handed over. In case of read, the content of the page does not make any meaning as it is accessed for the first time by the process. Anyway to avoid confusions kernel prefers to give a zeroed page to the process. Again the tricky demand paging algorithm finds out that there is no necessity to allocate a new page frame filled with zeroes just to READ and so allocates a global zeroed page (always residing at the 5th page frame). And sets the page table entry with the zero page's physical address (5th page frame) and marks it non-writable. (And that’s what i call resource management at its best).Only when the process wants to write to this page the kernel allocates a new page frame and follows the COW.

Copy - On - Write

Copy-On-Write is an interesting technique which has saved so much time and overhead, especially, during process creation. Whenever a child is created it has the same address space as the parent. Earlier this required heavy page frame allocations, since the whole address space was awkwardly duplicated and page frames were allocated for the child's entire process address space. The pain was felt when most of the children used to start execution by loading a new program thereby discarding the entire inherited address

space. To solve all these COW came into existence. The basic principle of COW is as follows,

"Never allocate a new page frame (for a page already in memory) to any process unless it is for WRITE purpose" explain in detail with an example, consider the child process. Whenever a child is forked To instead of creating page frames for the child the COW technique makes the page frames of the parent as sharable (only READ-able) pages. That saves a lot of page frames. When one process wants to write to that page, a new page frame has to be allocated (but not for the entire space) and kernel allocates a new page frame, copies the information from the sharable page and marks the new page writable for the process. The old sharable page is still sharable. When the next process wants to write to the page, the algorithm checks whether its the sole owner of the page and if it is , makes it writable else gets a new page frame and copies the information.

Generally when a bad area has been found we have seen above that the process is killed. But its not the same when the page fault occurs in the kernel mode. In kernel mode the page fault may be in a bad area because of two reasons. One is a wrong linear address has been passed to kernel as parameter. In that case the system call handler is terminated. The second reason is because of some kernel bug. In this case the CPU registers and kernel mode stack are dumped and the kernel comes to a halt.

This is how page fault exceptions are handled.

The Linux MMU uses some famous algorithms like buddy system and slab allocator for allocation of page frames. They are not dealt here as they are completely architecture independent and also fairly common algorithms for which many dedicated articles could be found.

Cache and TLB handling

The cache and TLB (Translation Look Aside Buffer) are completely hardware stuff and the Linux kernel has to take care of only one thing after it enables them, FLUSHING. The caches and TLBs are very sensitive and wrong entries in them may cause the kernel to go haywire in-spite of the meticulously planned algorithms. The nature of the cache (write-

through or copy-back) has to consider and the flushes have to be applied.

The places where the cache flushing is necessary is jotted below

1. At the time of page table creation (initialization phase, clear the junk in cache)
2. At the time of page faults when new page frames are allocated. Clearing is done only when the page frame is mapped
3. At the time of execve when a new page is mapped into an address space.
4. At the time of high-mem pages.

The TLBs also should be flushed at the appropriate time. An extensive documentation about the various functions that are used to flush the TLBs and cache is provided in the Linux/documentation/cachetlb.txt and is added at the source code's to be changed portion of teh document.

SWAPPING

Swapping is based on the idea of using the disk storage as an extension of RAM. This swapping gives extended address space and an expanded amount of RAM. To put in one line "copying of RAM page frames to disk and bringing them back when necessary is called swapping of page frames". This is useful when programs with very large data structures are run or large programs are run many times simultaneously. If swapping does not exist in such conditions the kernel will be forced to kill a user process (poor fellow) so that it can reclaim that page frame and give it to the new request for page frame. The only drawback of swapping is it takes a lot more time to access a swapped page as the disk access speeds are less than RAM by a whooping margin. In early days the swapping was done for entire process address space but nowadays it is done for pages (the smallest unit in memory management). Swapping can be done only for pages that belong to an anonymous memory region of a process, modified pages that belong to a private memory region and pages that belong to IPC shared memory region. The page to be swapped is done using LRU algorithms. Kernel pages and pages containing file data are never swapped. (For files the best swap is the file in disk itself) Generally the swap area is arranged in slots and each slot contains exactly one page. As

much as 32 (generally) swap areas can be had in a a system and swap areas can be files or disk partitions. Having different swap areas will reduce the pressure on one in case a lot of swap is happening. Swapping is mainly done as a last resort when the kernel is dangerously low in memory virtually suffocating for space. As a safety precaution the kernel keeps a small amount of reserved memory just in-case the memory goes so low that even the free pages code is unable to run.

When a page is swapped the swapping algorithm makes use of the page present flag in the page table entry. If a page is swapped to disk its corresponding page present bit is reset and the rest of the page table entry is used to hold the page identifier.

Swap Caches

Swap caches were implemented to avoid race conditions due to processes competing to access page frames that are being swapped. If only one process is addressing and using the memory then the race conditions do not exist and semaphores can be used to handle the situation. Whereas in case of many processes accessing the page frame through different page descriptors the race conditions are crucial. Due to race conditions a situation may arise when a page appears as swapped in one process and appears to be present in another.(well thats a very unique and painful) To avoid this swap caches are used. The swap caches have a collection of page frames which have been moved to the swap area space. Well the swap cache needs memory again isn't it? That has been handled cleverly. No separate data structures are involved in the swap cache. Instead certain pages are said to be in swap cache if certain fields are set. this swap cache tends to have the frame in memory until the page frame user count goes to zero (not used by any process) and does not allow the kernel to reclaim the page frame.

A cascade of functions are called and a definite process is followed when swapping out and swapping in of pages. The details about these functions are futile here as the swapping function is a cleverly implemented algorithm which is by and large independent of the architecture.

THE SOURCE CODE PORTION THAT MAY NEED MODIFICATION

The Linux Source is well abstracted into architecture dependent and independent part. The architecture dependent part is restricted to the arch directory in the Linux Source root directory. The MMU code resides mainly in mm directory at the top directory and arch/model/mm directory and their include files are in include/asm-model and include/Linux directories. The code in the top-level mm directory and include/Linux directory are fairly architecture independent high level code where as the arch/model/mm and include/asm-model are architecture specific code.

In the following piece the functions and the file names that were modified while porting Linux to model processor are given. Also a mention is made about all the functions and filenames inside the arch/model/mm directory(even that were not modified much) as it may help in porting any type of processor. While reading the code of any other processor (say Intel, arm, SPARC, etc.) the same function names and file names may not exist but the core functions will be more or less the same and the other functions may be implemented in a slightly different manner with a different name.

The following are the files that will/may need change in case of porting.

1. arch/model/kernel/head.S
2. arch/model/kernel/setup.c
3. arch/model/mm/fault.c
4. arch/model/mm/init.c
5. arch/model/memory.c
6. arch/model/kmap.c
7. arch/model/model.c

Include files

----

8. include/asm-model/model_pgalloc.h
9. include/asm-model/model/model_pgtable.h

10. include/asm-model/bootinfo.h
11. include/asm-model/cachectl.h
12 include/asm-model/cache.h
13. include/asm-model/mmu.h
14. include/asm-model/mmu_context.h
15. include/asm-model/page.h
16. include/asm-model/page_offset.h
17. include/asm-model/pgalloc.h
18. include/asm-model/pgtable.h
19. include/asm-model/shm.h
20. include/asm-model/shmbuf.h
21. include/asm-model/shmparam.h
22 include/asm-model/virtconvert.h

All the functions are analyzed here and a clue on the porting effort can be achieved. The files mentioned here and the functions may contain code pertaining to other parts of the architecture and kernel (interrupt, process handling, etc.) other than the MMU code. Those are omitted in the following discussion.

Head.S

This is the very first code that runs in the Linux kernel. And obviously it is completely in assembly. The initial part of the head.S code runs without any translation (MMU off). And this part has to set all the page tables for the initial kernel pages. All the access right flags and cache enable flags are set properly in the page table entries. Generally the assembly function mmu_map does this. And once the tables are set the MMU is switched ON. The respective registers are set with the appropriate values to switch ON MMU (generally mmu_engage assembly function does this). Once the MMU is switched ON the work of the MMU part in head.S is over.

Care has to be taken in setting up the page tables and also they should be dynamic. Dynamic setting of page tables mean that the there should be scope to increase the mapping size in head.S itself. To elaborate on this consider this memory map. The first page consists of one Page Global directory and 7 pointer tables. The next page contains 16 page tables.

*Understand that there are pages required in RAM to store the page tables itself. One page has 4096 bytes and can store only 8, 128-entry tables. (8 tables * 128 entries * 4byte long entry = 4096 Bytes). That's the reason for 1 global directory and 7 pointer tables. (Entries in Global dir, pointer table = 128). The page table has 64 entries and so a page can accommodate 16 page tables.*

The 3rd page is left free (safety reasons) and kernel starts from the 4th page and code runs for many pages. The address where the kernel code ends is stored in a variable pavialmem. Now in this idea if we have hard-coded the memory addresses where these PGD and PMD are mapped, then we are resisting ourselves to 16 * 64 * 4096 = 4MB of memory alone (Yes! where can we keep the 17th page table if we want more memory). Well! thats static. We are forced to map only 4 MB in head.S.

As an alternative way if we write the mmu_map code based on the pavailmem so that it will create PMD and page tables based on pavailmem then we will be able to map any amount of memory. Consider this revised map. In this the first page contains 1 PGD and 7 PMD, the next page is free and 3rd page kernel starts, the place where the kernel ends the page tables start and pavailmem will point to the last used page. So now the code can map any amount of memory and thats dynamic.

A rough algorithm is given here along with a figure.

1. Map PGD as the first table in PAGE 1
2. Map first PMD as second table in PAGE 2, Set PMD_counter = 2
3. Map first page table as first table in PAGE at pavailmem , set PT_counter = 1, increase pavailmem by one PAGE
4. Check PMD_counter = 8
   a: If YES get PAGE at current pavailmem and set PMD as the first table, reset PMD_counter = 1
   b: Increase pavailmem by one PAGE.
   c: If NO place the PMD in the page where the previous PMD was placed, next to it
5. Check PT_counter = 32
   a: If YES get PAGE at current pavailmem and set Page table as the first table, reset PT_counter = 1
   b: Increase pavailmem by one PAGE.
   c: If NO place the Page table in the page where the previous Page table was placed, next to it.

Thats the way the the mapping can be dynamic.

MAP -- STATIC

Page 1
Root Table
Pointer Table 1
Pointer Table 2
Pointer Table 3
Pointer Table 4
Pointer Table 5
Pointer Table 6
Pointer Table 7
Page 2
16 Page Tables
Page 3
EMPTY PAGE
Page 4
Kernel Start
Page N
Kernel End
pavailmem

MAP -- DYNAMIC

Page 1
Root Table
Pointer Table 1
Pointer Table 2
Pointer Table 3
Pointer Table 4
Pointer Table 5
Pointer Table 6
Pointer Table 7
Page 2
EMPTY PAGE
Page 3
Kernel Start
Page N
Kernel End
pavailmem
16 Page Tables
new pavailmem

Figure 5

setup.c file houses the following functions.

model_parse_bootinfo( )
setup_arch( )

The model_parse_bootinfo( )

This function is responsible for parsing the boot options, passed through by the boot-loader and the BIOS. Generally they have the details about the total memory and the layout of the memory in the physical address space. It has the size of memory that Linux can use(while booting we can request the Linux to use less memory). This may need tweaks while porting as the hardware target may not have a boot-loader or BIOS. In that case we may have to hard-code some of the facts or imply some other method to pass the information.

setup_arch()

This function actually calls the model_parse_bootinfo( ) and does the necessary calculations to find the exact amount of memory to be mapped, location etc. and calls the paging init() function. It also reserves all the pages(frames) . It also calls the paging_init() to complete the kernel mappings started in head.S. This function may need modifications in case we are interested in reserving certain memory specifically due to hardware design and also a few modifications corresponding to the bootinfo() modifications.

fault.c

This is the most important architecture specific file regarding the MMU .It contains the following functions

do_page_fault ( )
send_fault_sig ( )

do_page_fault ( )

The do_page_fault ( ) function is the page fault handler explained in the Linux MMU portion, earlier. The decision whether the faulting address is in good area or in bad area based on error codes is fairly architecture based. Each architecture will have its own way to send the reason for the fault and each one has its own meaning. For example Intel uses a 3 bit error code, whereas Model uses a 2 bit error code. This error register part of the code has to be designed strictly according to the architecture. Other than this the rest of the handler does not require mentionable change generally. Anyway as the algorithm of fault handling is explained in detail earlier any change that may be required for an architecture with a very unorthodox type of MMU can also be made with little effort.

send_fault_sig ( )

This function takes care of the proceedings in case of bad area. The bad user faults and kernel faults are dealt here. Process killed for user and process killed or kernel halted in case of kernel. This can be changed if the architectures demand a different way of handling the user/kernel faults.

init.c

The function that interests our study is mem_init ()
Apart from this the file also contains functions like show_mem(), show_freepages, si_meminfo, free_initrd_mem() which have self explanatory names and does not attract much if not any work while porting. The zeroed page discussed in Copy On Write is also initialized here.

mem_init()

The mem_init as the name suggests initializes the memory for the kernel and the user processes. All the pages reserved by bootmem function are freed in this function and the pages that has to be reserved are reserved (like kernel code, data and page tables). All

the code, data, init and other pages are calculated here. Little or no porting effort is required in this function.

memory.c

The functions in this file includes pointer table related functions like get_pointer_table, init_pointer_table and free_pointer_table. Care has to be taken when porting regarding the size and no. of entries on the pointer table. The coding may slightly vary in this regard.

This file also contains the function mm_vtop( ) and mm_ptov( ) which are 100% architecture dependent. The virtual to physical (mm_vtop) and physical to virtual (mm_ptov) are dependent on the architecture and the memory map we have set.

kmap.c

This file houses all the IO mapping functions like get_io_area, free_io_area, io_remap and io_unmap

We do not have I/O accesses in Model and so are many processors. So we go in for memory mapped I./O. The get_io_area and free_io_area are functions similar to functions used for getting virtual memory areas (get_vm_area) in case the I/O area is page size> if I/O area is lesser than page size then the function has to split a page and offer and so a bit complicated.

io_remap( ) function resembles mapping a chunk of physical memory (map_chunk). It maps a physical address piece onto the kernal address space. Again the memory map that has been decided for this platform comes into picture while porting is concerned.

model.c

This file contains model processor dependent code. The functions included are paging_init( ), map_chunk( ).

The paging_init( ) function, finishes the kernel mapping started by the head.S code.This function in-turn calls the map_chunk() with proper parameters obtained in setup_arch( ) to create the mapping for Kernel address space.The map chunk() function employsthe kernel_pg_table(),kernel_ptr_table( ) and other small functions which help in setting the page table, pointer table etc.These small functions should be carefully coded according to the page table architecture. Of the include files the following files worth a description .All others are just definitions that have to be changed if need be so.

model_pgtable.h

This file has macro definitions related to the page table. All the page flag macros are defined here. Also small macros for setting the value of pointer table entry with the page table base,etc. and offset calculating macros are defined here. This file needs attention when porting as we may have to re-write these macros.

model_pgalloc.h

This file has macro definitions related to the page allocation. Macros related to allocations like pmd allocation, pgd allocation and freeing of them are defined. Also all TLB flushing inline functions are defined here. These functions need to be implemented.

pgalloc.h

This file has all the cache flush related inline functions. The file contains different codes for copy-back and write-through mode of cache. The necessary functions have to be chosen based on the hardware and has to be implemented.

virtconvert.h

This file has functions supporting the virtual to physical conversion and vice versa. These functions have to be implemented. The **cache flush functions**' (defined in pgalloc.h , model_pgalloc.g and other files) importance (derived from

Linux/Documentation/cachetlb.txt) are given below separately for sake of convenience.First, the TLB flushing interfaces, since they are the simplest. The "TLB" is abstracted under Linux as something the CPU uses to cache virtual-->physical address translations obtained from the software page tables. Meaning that if the software page tables change, it is possible for stale translations to exist in this "TLB" cache. Therefore when software page table changes occur, the kernel will invoke one of the following flush methods _after_ the page table
changes occur:

1) void flush_tlb_all(void)

The most severe flush of all. After this interface runs, any previous page table modification whatsoever will be visible to the CPU. This is usually invoked when the kernel page tables arechanged, since such translations are "global" in nature.

2) void flush_tlb_mm(struct mm_struct *mm)

This interface flushes an entire user address space fromthe TLB. After running, this interface must make sure that any previous page table modifications for the address space 'mm' will be visible to the CPU. That is, after running, there will be no entries in the TLB for 'mm'. This interface is used to handle whole address space page table operations such as what happens during fork, and exec.

3) void flush_tlb_range(struct mm_struct *mm,unsigned long start, unsigned long end)
Here we are flushing a specific range of (user) virtual address translations from the TLB. After running, this interface must make sure that any previous page table modifications for the address space 'mm' in the range 'start' to 'end' will be visible to the CPU. That is, after running, there will be no entries in the TLB for 'mm' for virtual addresses in the range 'start' to 'end'. Primarily, this is used for munmap() type operations.The interface is provided in hopes that the port can find a suitably efficient method for removing multiple page sized translations from the TLB, instead of having the kernel call flush_tlb_page

(see below) for each entry which may be modified.

4) void flush_tlb_page(struct vm_area_struct *vma, unsigned long page)

This time we need to remove the PAGE_SIZE sized translation from the TLB. The 'vma' is the backing structure used by Linux to keep track of mmap'd regions for a process, the address

space is available via vma->vm_mm. Also, one may test (vma->vm_flags & VM_EXEC) to see if

this region is executable (and thus could be in the 'instruction TLB' in split-tlb type setups).

After running, this interface must make sure that any previous page table modification for address space 'vma->vm_mm' for user virtual address 'page' will be visible to the CPU. That

is, after running, there will be no entries in the TLB for 'vma->vm_mm' for virtual address 'page'. This is used primarily during fault processing.

5) void flush_tlb_pgtables(struct mm_struct *mm,unsigned long start, unsigned long end)

The software page tables for address space 'mm' for virtual addresses in the range 'start' to 'end' are being torn down.

Some platforms cache the lowest level of the software page tables in a linear virtually mapped array, to make TLB miss processing more efficient. On such platforms, since the TLB is caching the software page table structure, it needs to be flushed when parts of the software page table tree are unlinked/freed.
Sparc64 is one example of a platform which does this.

Usually, when munmap()'ing an area of user virtual address space, the kernel leaves the page table parts around and just marks the individual pte's as invalid. However, if very large portions of the address space are unmapped, the kernel frees up those portions of the software page tables to prevent potential excessive kernel memory usage caused by

erratic mmap/mmunmap sequences.

It is at these times that flush_tlb_pgtables will be invoked.

6) void update_mmu_cache(struct vm_area_struct *vma, unsigned long address, pte_t pte)

At the end of every page fault, this routine is invoked to tell the architecture specific code that a translation described by "pte" now exists at virtual address "address" for address space "vma->vm_mm", in the software page tables.

A port may use this information in any way it so chooses.For example, it could use this event to pre-load TLB translations for software managed TLB configurations.The sparc64 port currently does this.

Next, we have the cache flushing interfaces. In general, when Linux is changing an existing virtual-->physical mapping to a new value, the sequence will be in one of the following forms:

1) flush_cache_mm(mm);
   change_all_page_tables_of(mm);
   flush_tlb_mm(mm);

2) flush_cache_range(mm, start, end);
   change_range_of_page_tables(mm, start, end);
   flush_tlb_range(mm, start, end);

3) flush_cache_page(vma, page);
   set_pte(pte_pointer, new_pte_val);
   flush_tlb_page(vma, page);

The cache level flush will always be first, because this allows us to properly handle systems whose caches are strict and require a virtual-->physical translation to exist for a virtual address

when that virtual address is flushed from the cache. The HyperSparc CPU is one such CPU with this attribute.

The cache flushing routines below need only deal with cache flushing to the extent that it is necessary for a particular CPU. Mostly, these routines must be implemented for cpus which have virtually indexed caches which must be flushed when virtual-->physical translations are changed or removed. So, for example, the physically indexed physically tagged caches of IA32 processors have no need to implement these interfaces since the caches are fully synchronized and have no dependency on translation information.

Here are the routines, one by one:

1) void flush_cache_all(void)

The most severe flush of all. After this interface runs, the entire CPU cache is flushed. This is usually invoked when the kernel page tables are changed, since such translations are "global" in nature.

2) void flush_cache_mm(struct mm_struct *mm)

This interface flushes an entire user address space from the caches. That is, after running, there will be no cache lines associated with 'mm'.

This interface is used to handle whole address space page table operations such as what happens during fork, exit, and exec.

3) void flush_cache_range(struct mm_struct *mm,unsigned long start, unsigned long end)

Here we are flushing a specific range of (user) virtual addresses from the cache. After running, there will be no entries in the cache for 'mm' for virtual addresses in the range 'start' to 'end'.

Primarily, this is used for munmap() type operations.

The interface is provided in hopes that the port can find a suitably efficient method for

removing multiple page sized regions from the cache, instead of having the kernel call flush_cache_page (see below) for each entry which may be modified.

4) void flush_cache_page(struct vm_area_struct *vma, unsigned long page)

This time we need to remove a PAGE_SIZE sized range from the cache. The 'vma' is the backing structure used by Linux to keep track of mmap'd regions for a process; the address space is available via vma->vm_mm. Also, one may test (vma->vm_flags & VM_EXEC) to see if this region is xecutable (and thus could be in the 'instruction cache' in "Harvard" type cache layouts).

After running, there will be no entries in the cache for 'vma->vm_mm' for virtual address 'page'. This is used primarily during fault processing.

There exists another whole class of CPU cache issues which currently require a whole different set of interfaces to handle properly. The biggest problem is that of virtual aliasing in the data cache of a processor.

Is your port susceptible to virtual aliasing in its D-cache? Well, if your D-cache is virtually indexed, is larger in size than PAGE_SIZE, and does not prevent multiple cache lines for the same physical address from existing at once, you have this problem.

If your D-cache has this problem, first define asm/shmparam.h SHMLBA properly, it should essentially be the size of your virtually addressed D-cache (or if the size is variable, the largest possible size). This setting will force the SYSv IPC layer to only allow user processes to mmap shared memory at address which are a multiple of this value.

NOTE: This does not fix shared mmaps, check out the sparc64 port for one way to solve this (in particular SPARC_FLAG_MMAPSHARED).

Next, you have two methods to solve the D-cache aliasing issue for all other cases. Please keep in mind that fact that, for a given page mapped into some user address space, there is always at least one more mapping, that of the kernel in its linear mapping

starting at PAGE_OFFSET. So immediately, once the first user maps a given physical page into its address space, by implication the D-cache aliasing problem has the potential to exist since the kernel already maps this page at its virtual address.

First, I describe the old method to deal with this problem. I am describing it for documentation purposes, but it is deprecated and the latter method I describe next should be used by all new ports
and all existing ports should move over to the new mechanism as well.

flush_page_to_ram(struct page *page)

The physical page 'page' is about to be place into the user address space of a process. If it is possible for stores done recently by the kernel into this physical page, to not be visible to an arbitrary mapping in user space, you must flush this page from the D-cache.

If the D-cache is write back in nature, the dirty data (if any) for this physical page must be written back to main memory before the cache lines are invalidated. Admittedly, the author did not think very much when designing this interface. It does not give the architecture enough information about what exactly is going on, and there is no context to base a judgment
on about whether an alias is possible at all. The new interfaces to deal with D-cache aliasing are meant to address this by telling the architecture specific code exactly which is going on at the proper points in time.

Here is the new interface:

Void copy_user_page(void *to, void *from, unsigned long address)

Void clear_user_page(void *to, unsigned long address)

These two routines store data in user anonymous or COW pages. It allows a port to efficiently avoid D-cache alias issues between user space and the kernel. For example, a port may temporarily map 'from' and 'to' to kernel virtual addresses during the copy. The

virtual address for these two pages is chosen in such a way that the kernel load/store instructions happen to virtual addresses which are of the same "color" as the user mapping of the page. Sparc64 for example, uses this technique.

The "address" parameter tells the virtual address where the user will ultimately this page mapped. If D-cache aliasing is not an issue, these two routines may simply call memcpy/memset directly and do nothing more.

void flush_dcache_page(struct page *page)

Any time the kernel writes to a page cache page, _OR_ the kernel is about to read from a page cache page and user space shared/writable mappings of this page potentially exist, this routine is called.

NOTE: This routine need only be called for page cache pages which can potentially ever be mapped into the address space of a user process. So for example, VFS layer code handling vfs symlinks in the page cache need not call this interface at all.

The phrase "kernel writes to a page cache page" means, specifically, that the kernel executes store instructions that dirty data in that page at the page->virtual mapping of that page. It is important to flush here to handle D-cache aliasing, to make sure these kernel stores are
visible to user space mappings of that page.

The corollary case is just as important, if there are users which have shared+writable mappings of this file, we must make sure that kernel reads of these pages will see the most recent stores done by the user. If D-cache aliasing is not an issue, this routine may simply be defined as a nop on that architecture.

There is a bit set aside in page->flags (PG_arch_1) as "architecture private". The kernel guarantees that, for pagecache pages, it will clear this bit when such a page first enters the pagecache.

This allows these interfaces to be implemented much more efficiently. It allows one to

"defer" (perhaps indefinitely) the actual flush if there are currently no user processes mapping this page. See sparc64's flush_dcache_page and update_mmu_cache implementations for an example of how to go about doing this.

The idea is, first at flush_dcache_page() time, if page->mapping->i_mmap{,_shared} are empty lists, just mark the architecture private page flag bit. Later, in update_mmu_cache(), a check is made of this flag bit, and if set the flush is done and the flag bit is cleared.

void flush_icache_range(unsigned long start, unsigned long end)

When the kernel stores into addresses that it will execute out of (eg when loading modules), this function is called.

If the icache does not snoop stores then this routine will need to flush it.
void flush_icache_page(struct vm_area_struct *vma, struct page *page) All the functionality of flush_icache_page can be implemented in flush_dcache_page and update_mmu_cache. In 2.5 the hope is to remove this interface completely.

Conclusion

Once having known the answers to the basic questions a clear plan has to be devised to port step by step. Memory is the most important for the operating system and it issues related to that should be resolved even before other modules in the kernel could run. A memory layout (where the page tables, kernel, kernel data, etc would reside.) has to be decided upon. The place where the initial kernel page tables are to be set and the size for which it has to be set has to be planned and switching the MMU ON is based on this. Once these are planned and coding is complete there will be a minimum place where the kernel can reside and run. Now the rest of the code is ported based on the hardware. Don't forget to have the hardware table prepared near you.

HAPPY PORTING!!