The Linux Kernel's Memory Management Unit API

William Gatliff

Table of Contents

Overview

This paper introduces the Linux 2.4 kernel's memory management hardware API, from an embedded developer's perspective. It briefly discusses the fundamental concepts of memory management, and how Linux uses the host environment's memory management hardware to implement memory services like virtual, shared and protected memory.

This article then focuses on the programming interface Linux uses to interact with the host processor's Memory Management Unit (MMU).

What is Memory Management?

The term *memory management* refers to the mechanisms implemented by an operating system to provide memory-related services to applications. These services include *virtual memory* (use of a hard disk or other non-RAM storage media to provide additional program memory), *protected memory* (exclusive access to a region of memory by a process), and *shared memory* (cooperative access to a region of memory by multiple processes).

Linux's memory management services are built on a programming foundation that includes a peripheral device called a *Memory Management Unit (MMU)*. An MMU translates physical memory addresses to *linear addresses* used by the operating system, and requests a *page fault* interrupt when the CPU tries to access memory that it is not entitled to.

Page Directories and Page Tables

The fundamental unit of memory under Linux is the *page*, a nonoverlapping region of contiguous memory. All available physical memory is organized into pages near the end of the kernel's boot process, and pages are doled out to and revoked from processes by the kernel's memory management algorithms at runtime. Linux uses 4k-byte pages for most processors.

Under Linux, each process has an associated *page directory* and *page table*, data structures that keep track of which memory pages are being used by the process, and what their virtual and physical addresses are. When a process needs more memory, an unused page is taken from a global list called `pgdata_list`, and added to the process's page table. When the process exits, its pages are returned to `pgdata_list`.

Mapping a Page

When a memory page is given to a process, the page is said to have been *mapped* to that process. To map a page to a process, the MMU's configuration is adjusted so that the physical address of that page (the number that actually goes out on the processor's address lines) is translated to the *linear* address needed by the process.

Memory pages are usually scattered around in memory; the linear-to-physical mapping provided by the MMU makes the pages appear to have contiguous addresses to the operating system.

Page Faults

When a process tries to access memory in a page that is not known to the MMU, the MMU generates a *page fault* exception. The page fault exception handler examines the state of the MMU hardware and the currently running process's memory information, and determines whether the fault is a "good" one, or a "bad" one. Good page faults cause the handler to give more memory to the process; bad faults cause the handler to terminate the process.

Good page faults are expected behavior, and occur whenever a program allocates dynamic memory, runs a section of code or writes a section of data for the first time, or increases its stack size. When the process attempts to access this new memory, the MMU declares a page fault, and the OS adds a fresh page of memory to the process's page table. The interrupted process is then resumed.

Bad faults occur when a process follows a NULL pointer, or tries to access memory that it doesn't own. Bad faults can also occur due to programming bugs in the kernel, in which case the handler will print an "oops" message before terminating the process.

The `do_page_fault()` Function

`Do_page_fault()` is the Linux kernel's page fault interrupt handler. Although generally consistent between kernel versions, its implementation is found in the processor specific portions of the kernel source tree. The Hitachi SH version is in the file `arch/sh/mm/fault.c`.

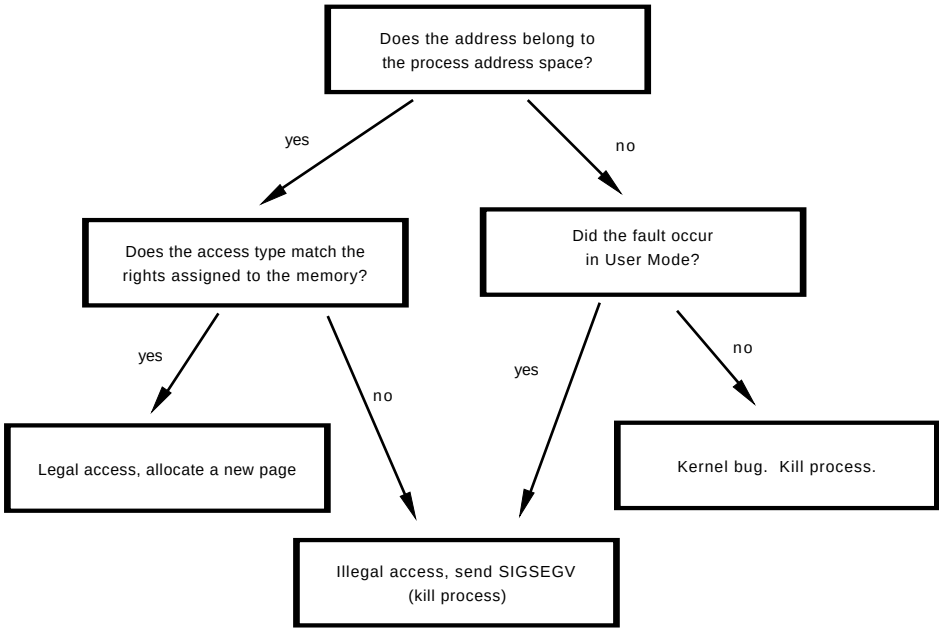
The `do_page_fault()` function takes three parameters:

- A pointer to a pt_regs structure, which contains the values of microprocessor registers when the page fault occurred.
- An error code that indicates the reason for the page fault.
- The address that generated the page fault.

The error code tells `do_page_fault()` if the process was reading from or writing to the associated address, and whether the page being accessed was actually in memory (vs. stored on disk) when the fault occurred. `Do_page_fault()` combines this with information in the process's mm_struct structure, to determine if the process has legitimate rights to access the memory address or not.

The logic applied to the access by `do_page_fault` is summarized in the flowchart shown in Figure 1. In summary, if the process has rights to the address then a new memory page is mapped to that address and the interrupted process is resumed. If the process does not have rights to the address, the process is terminated and a *segmentation fault* error is reported to the parent process.

Figure 1. The `do_page_fault()` function.



The `pte_*()` Functions

Linux's page fault handler uses the kernel's memory management hardware API to manipulate page directories and page tables. Most of these functions begin with the name `pte_*`, which stands for *Page Table Entry*. There are at least fifteen of these functions (the number depends on the kernel version), including versions that map and unmap pages, adjust the access rights for a page, and mark pages as written or unwritten.

The `pte_*()` functions interact with MMU hardware, and their implementations are therefore highly hardware-specific. They are defined in the architecture-specific portions of the kernel source tree. The Hitachi SH versions are defined in `include/asm-sh/pgtable.h` and `include/asm-sh/pgtable-2level.h`.

Although it varies with kernel versions, the `pte_*()` functions are usually built on just two base functions called `set_pte()` and `pte_val()`. Here is the Hitachi SH4's implementation for `pte_clear()`, which clears a page table entry by calling `set_pte()` with a zero argument:

```
typedef struct {unsigned long pte;} pte_t;
#define __pte(x) ((pte_t) { (x) } )
#define pte_clear(xp) do { set_pte(xp, __pte(0)); } while (0)
```

Here is the definition for `pte_mkdirty()`. This macro uses `set_pte()` to set the Hitachi SH4 MMU's `D` bit in the requested page table entry. This marks the page as active. Linux's virtual memory manager will not move an active memory page to disk until all of a process's inactive pages have been moved.

```
// The page table entry's D bit; see section 3.4, Memory Management
// Unit, of the Hitachi SH4 manual.
#define _PAGE_DIRTY 0x004

static inline pte_t pte_mkdirty(pte_t pte)
{
   set_pte(&pte, __pte(pte_val(pte) | _PAGE_DIRTY));
   return pte;
}
```

The `pte_present()` function queries the `V` bit in the requested page table entry, to see if the requested page is in memory or has been moved to disk.

```
// V bit; page is valid
#define _PAGE_PRESENT   0x100

// a software-defined bit; this bit gets masked
// before the pte is physically written to the MMU
#define _PAGE_PROTNONE  0x200
```

```
#define pte_present(x) (pte_val(x) & (_PAGE_PRESENT | _PAGE_PROTNONE))
```

Once a page table entry is properly configured, it is written to the MMU using the `flush_tlb_page()` function. This function is called by a sub-function of `do_page_fault()`, near the end of the page fault handling process.

The `flush_tlb_*()` Functions

When a page table entry structure is configured, it must be written to the MMU in order for its settings to take affect. The functions that do this are named `flush_tlb_*()`, i.e. `flush_tlb_page()`, `flush_tlb_range()`, and so forth. The TLB in the name stands for *Translation Lookaside Buffer*, the traditional name used for an MMU's internal memory mapping registers.

For some processors like the Hitachi SH4, manipulating the MMU state is a two-step process. The `flush_tlb_*()` functions manage the first step, which is to update the MMU's memory-mapped page data arrays. Once these arrays are updated, an MMU-specific opcode is emitted by `update_mmu_cache()` to actually commit the new MMU configuration to the MMU hardware. `Update_mmu_cache()` is called at the end of page fault handling.

The code in Figure 2 is a partial listing of `flush_tlb_page()` for the Hitachi SH4. The code appears in `arch/sh/mm/fault.c` in the kernel source tree.

Figure 2. The `flush_tlb_page()` function.

```
void flush_tlb_page(struct vm_area_struct *vma, unsigned long page)
{
  if (vma->vm_mm && vma->vm_mm->context != NO_CONTEXT) {
    unsigned long flags;
    unsigned long asid;
    unsigned long saved_asid = MMU_NO_ASID;

    asid = vma->vm_mm->context & MMU_CONTEXT_ASID_MASK;
    page &= PAGE_MASK;

    save_and_cli(flags);
    if (vma->vm_mm != current->mm) {
      saved_asid = get_asid();
      set_asid(asid);
    }

    addr = MMU_UTLB_ADDRESS_ARRAY | MMU_PAGE_ASSOC_BIT;
    data = page | asid;
```

```
    ctrl_outl(data, addr);

    if (saved_asid != MMU_NO_ASID)
      set_asid(saved_asid);
      restore_flags(flags);
  }
}
```

The `update_mmu_cache()` Function

The `update_mmu_cache()` adjusts MMU hardware in response to a page table entry adjustment performed by the `pte_*()` and `flush_tlb_*()` functions. The code for the Hitachi SH4 version is shown in Figure 3.

The Hitachi SH4 version of `update_mmu_cache()` checks the page table entry for validity, loads the `PTEH`, `PTEA`, and `PTEL` registers, then runs the `ldtlb` opcode to commit the register settings to the MMU's translation lookaside buffers.

Figure 3. The `update_mmu_cache()` function.

```
void update_mmu_cache(struct vm_area_struct * vma,
  unsigned long address, pte_t pte)
{
  unsigned long flags;
  unsigned long pteval;
  unsigned long vpn;
  struct page *page;
  unsigned long ptea;

  /* Ptrace may call this routine. */
  if (vma && current->active_mm != vma->vm_mm) return;

  page = pte_page(pte);
  if (VALID_PAGE(page) && !test_bit(PG_mapped, &page->flags)) {
    unsigned long phys = pte_val(pte) & PTE_PHYS_MASK;
    __flush_wback_region((void *)P1SEGADDR(phys), PAGE_SIZE);
    __set_bit(PG_mapped, &page->flags);
  }

  save_and_cli(flags);

  /* Set PTEH register */
```

```
  vpn = (address & MMU_VPN_MASK) | get_asid();
  ctrl_outl(vpn, MMU_PTEH);

  pteval = pte_val(pte);
  /* Set PTEA register */
  ptea = ((pteval >> 28) & 0xe) | (pteval & 0x1);
  ctrl_outl(ptea, MMU_PTEA);

  /* Set PTEL register */
  pteval &= _PAGE_FLAGS_HARDWARE_MASK;
  ctrl_outl(pteval, MMU_PTEL);

  /* Load the TLB */
  asm volatile("ldtlb": /* no output */ : /* no input */ : "memory");
  restore_flags(flags);
}
```

Conclusion

Put simply, Linux's `pte_*()`, `flush_tlb_*()`, and `update_mmu_cache()` functions are the kernel's Memory Management Unit API. These functions connect Linux's hardware-generic memory management service algorithms to the host processor's Memory Management Unit hardware.

The algorithms that implement Linux's memory management services are complex and subject to constant revision. They are sufficiently abstract, however, in that they depend completely on the MMU API. A thorough understanding of this API is therefore essential to successful use of Linux in an embedded setting.

Copyright

About the Author

Bill Gatliff is an independent consultant with almost ten years of embedded development and training experience. He specializes GNU-based embedded development, and in using and adapting GNU tools to meet the needs of difficult development problems. He welcomes the opportunity to participate in projects of all types.

Bill is a Contributing Editor for Embedded Systems Programming Magazine (http://www.embedded.com/), a member of the Advisory Panel for the Embedded Systems Conference (http://www.esconline.com/), maintainer of the Crossgcc FAQ, creator of the gdbstubs (http://sourceforge.net/projects/gdbstubs) project, and a noted author and speaker.

Bill welcomes feedback and suggestions. Contact information is on his website, at http://www.billgatliff.com.