Memory Management

i386 Hardware Support
and
Linux Implementation

Agenda

- The LRU policy
- The MMU
- i386 page table structure
- i386 TLB
- Linux memory management

LRU Algorithm

- LRU associate with each page the time of that page's last use
- In LRU we will replace the page that has ***not been used for*** the longest period of time

7	0	1	2	0	3	0	4	2	3	0
7	7	7	2		2		4	4	4	0
	0	0	0		0		0	0	3	3
		1	1		3		3	2	2	2

LRU (Cont …)

- The LRU policy is considered to be quite good
- The major problem is how to implement LRU
- A good implementation may need substantial hardware assistance

Two implementations are feasible:

- **Counters:** Associate with each page table entry a time-of-use field. We replace the page with the smallest time value (This requires a search)
- **Stack:** Keep a stack of page numbers. When ever a page is referenced, it is removed from the stack and put on the top (A doubly linked list better than true stack)

The MMU

- The memory management system relies on the hardware to perform several tasks
- These tasks are performed by the **Memory Management Unit (MMU)**
- The primary task of the MMU is the translation of virtual addresses
- Most systems implement address translation using page tables
- Typically there is one page table for kernel addresses and one or more page tables for every process

Failing to translate

- Address translation **may fail** for tree reasons:
  - Bounds error – The address does not lie in the range of valid addresses for the process
  - Validation error – The page table entry is marked as invalid (can be used to simulate the referenced bit)
  - Protection error – The page does not permit the type of access desired
- In all such cases, the MMU raises an exception and passes control to the a handler in the kernel. Such and exception is called ***page fault***
- The handler may try to service the fault (bringing the page to memory
- Or notify the process by sending ***SIGSEGV*** Signal

x86 Memory management

- 4-gigabyte address space (32 bit addresses)
- Page size of 4096 bytes (4K)
- Support for both segmentation and paging
- Each virtual address consist of a 16 bit segment selector and 32 bit offset
- The segmentation layer converts this to a 32-bit linear address
- This is further translated to a physical address by the paging layer
- Paging may be disabled by clearing the high-order bit of the control register **CR0**

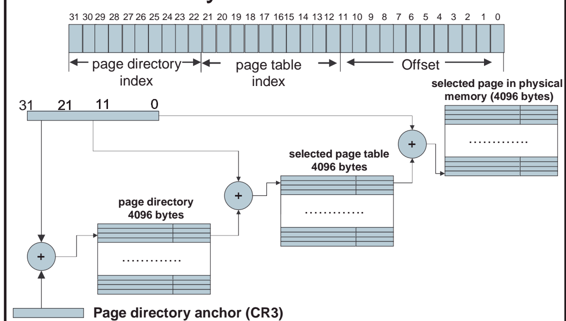
Translation Tables

- The translation tables are arranged in two level hierarchy
- The root level table is the **page directory table,** containing 1024 entries
- Each entry points to a subordinate table. Therefore up to 1024 subordinate tables are supported **(page table)**
- Hierarchy avoid reserving a large contiguous range of linear addresses
- It also support sparse mapping trees, where only several entries of the page directory point to page tables
- Control register 3, CR3, anchors the table structure

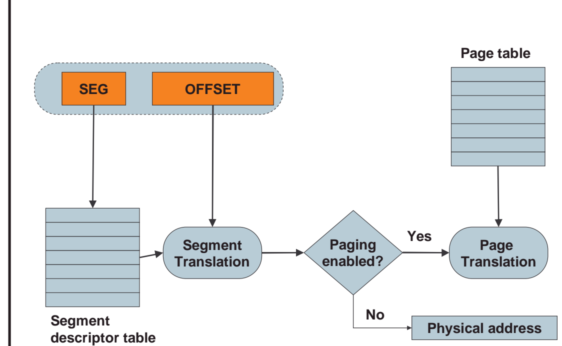
x86 memory management

- The process address space may contain up to 8192 segments
- Each segment is described by a segment descriptor (base, size, protection)
- Each process has its own local descriptor table (LDT)
  - Call gate segment (for system calls)
  - task state segment (TSS) for saving registers in context switch
- There is system wide global descriptor table (GDT) (kernel code, data and stack segments
- When translating the MMU uses the segment selector to identify the correct segment descriptor

Translating virtual address to Physical address



Address Translation



Page Directory Entry

- **Page table Address** defines the upper 20 bits of a page table physical memory
- **For OS Use** are 3 bits. The 80386 will never automatically alter these bits
- **D (Dirty)** indicate whether any of the 1024 pages of the page table has been written to.
- **A (Accessed)** indicates whether at least one of the 1024 pages has been read/written to
- **U/S (User/Supervisor)** protection
- **R/W (Read/Write)** protection
- **P (Present)** indicate whether the page present in physical memory

Page Protection

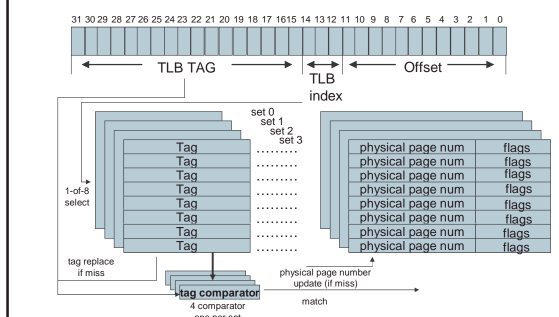
- The U/S and R/W bits in a page-directory entry are used to provide protection attributes for all the pages in the page table
- The U/S and R/W bits in the page-table entry provide similar protection for a page
- Both levels are used when paging occurs
- The more restrictive of the two apply

U/S	R/W	Permitted access from User Level	Permitted access from Supervisor level
0	0	None	Read and Write
0	1	None	Read and Write
1	0	Read Only	Read and Write
1	1	Read and Write	Read and Write

The i386 TLB

- The on chip TLB of the 800386 holds 32 translated-page values (covers 128k of memory range)
- Anytime a virtual address is within one of the 32 virtual pages the TLB can immediately supply the corresponding physical-page
- The TLB is managed as four sets of eight entries each
- This arrangement is called four-way set associativity
- For example when a TLB lookup occurs, each set is simultaneously index by the tree least-significant bits [14, 13 and 12] of the virtual page number
- The indexed entries are simultaneously compared with the current virtual address

80386 TLB Organization



i386 TLB (cont …)

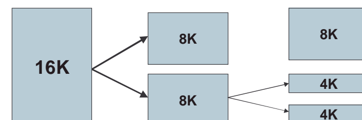
- The TLB in the x86 architecture is never directly accessed by the kernel
- The entire TLB is flushed whenever the page-directory-base-register is written to
  - directly by a move instruction
  - by context switching

Linux Memory Management

- Memory management under Linux has 2 parts:
  - Allocating and freeing physical memory
  - Handling virtual memory
- The code of the memory management system reside inside:
  - mm/
  - arch/i386/mm

Management of Physical Memory (the page allocator)

- The primary physical memory manager is the **page-allocator** *(file: mm/page_alloc.c)*
- Responsible for allocating and freeing all physical pages
- Uses the **buddy-heap algorithm** to keep track of available physical memory



Physical memory management (cont)

- Ultimately, all kernel memory allocation occur either statically (drivers) or dynamically by the page allocator
- Kernel functions can use other memory management subsystems to allocate memory:
  - The kmalloc allocator
  - The virtual memory system
  - The kernel caches: buffer and page cache

Virtual –Memory Regions

- There are several types of virtual memory regions, characterized by 2 properties:
- **The region backing store**:
  describe from where the pages for a region came. Most memory regions are backed by files or by nothing (demand-zero memory)
- **Reaction to writers**:
  the mapping of a region can be either private or shared.
  if a process writes to a privately mapped region the pager detects that a *copy-on-write* is necessary
  if a process writes to a shared region the shared object is updated

kmalloc() *(file mm/slab.c)*

- kmalloc() is used when the size is not known in advance and can be very small (few bytes)
- kmalloc() is safe from interrupts
- One of its parameters is request priority
  - GFP_USER – Allocate memory on behalf of a user (May sleep)
  - GFP_KERNEL – Allocate normal kernel ram (May sleep)
  - GFP_ATOMIC – Allocation will not sleep
- Interrupt routines passes the GFP_ATOMIC priority that guaranteed that the request is satisfied immediately or fail

Lifetime of a Virtual Address Space

- The kernel create new address space when:
- **process call exec()**, then a new empty address space is created. It is up to the loader routines to load the program
- **process call fork()**, creating a complete copy of the father address space
  - the vm_area_struct descriptors are copied
  - The parents page tables are copied, thus after the fork the parent and child share the same physical pages
- When the copy operation copy private pages, they are set as read-only and marked for ***copy-on-write***

Virtual Memory Views

- The virtual memory manager maintains 2 views of a process's address space
- **logical-view:** the address space consist of nonoverlapping regions, each represent a continuous, paged aligned subset of the address space.
  - Described by the *vm_area_struct*
  - The regions of each address space are linked into a balanced binary tree to allow fast lookup
- **physical view:** stored in the hardware page tables. The hardware tables determine the exact location of every page.

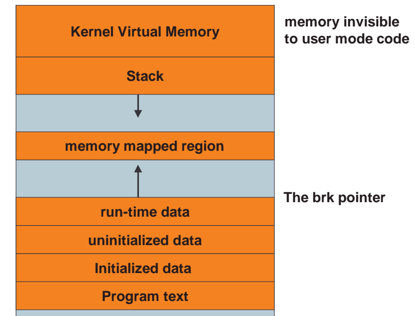
Swapping and Paging

- An important task of a virtual-memory system is to relocate pages from physical memory out to disk
- Early UNIX systems **swapped out the entire process** at once
- Modern UNIX systems relay more on paging
- In order to do that Linux too use the paging mechanism
- The paging system can be divided to the policy algorithm and the paging mechanism
- Linux's pageout policy is LFU (least frequently used)

Swapping and Paging

- The paging mechanism support both paging to dedicated swap devices and to files
- Blocks are allocated from the swap device according to bitmap of used blocks
- The allocator uses the next fit algorithm
- When a page is swapped out the page-not-present bit is set (or the present is unset)
- **cat /proc/swaps**
  shows information about the used swap devices

Memory layout of ELF programs



/proc/$pid/maps

- This file contain the currently mapped memory regions and their access permissions

```
[root@mos214 proc]# cat 1/maps
address           perm offset   dev   inode
08048000-0804e000 r-xp 00000000 03:01 38859      /sbin/init
0804e000-08050000 rw-p 00005000 03:01 38859      /sbin/init
08050000-08054000 rwxp 00000000 00:00 0
40000000-40016000 r-xp 00000000 03:01 71402      /lib/ld-2.2.4.so
40016000-40017000 rw-p 00015000 03:01 71402      /lib/ld-2.2.4.so
40017000-40018000 rwxp 00000000 00:00 0
40031000-40168000 r-xp 00000000 03:01 116282     /lib/i686/libc-
   2.2.4.so
40168000-4016d000 rw-p 00136000 03:01 116282     /lib/i686/libc-
   2.2.4.so
4016d000-40172000 rw-p 00000000 00:00 0
bfffe000-c0000000 rwxp fffff000 00:00 0
```

Execution and Loading of User programs

- When exec() is called the kernel invokes the loader routine to load the content of the program file into physical memory
- There is no single routine for loading but a table of routines
- Linux support at least 2 formats of executables
  - The old a.out format
  - ELF format
- The command “file” can tell you the file type, including the binary format (if the file is binary)

Mapping of Programs into Memory

- The pages of the binary file are mapped into regions of virtual memory
- Only when the program tries to access a given page a page fault will occur and the content of the file will be loaded to physical memory
- An ELF format consists of a header followed by several paged-aligned regions
- The ELF loader read the header map the sections of the file to separate regions
- In a reserved region at one end of the address space sits the kernel (inaccessible to users)