Introduction to Linux, for Embedded Engineers

Tutorial on Virtual Memory

Feb. 22, 2007

Tetsuyuki Kobayashi
Aplix Corporation
Target Audience of this Presentation

• People who have been engaged in projects on embedded devices, and who are now using Linux as operating system
Goals of this Presentation

• To understand the mechanism of virtual memory in Linux, and to make use of it for the current project
  – Although programs work without understanding the mechanism, it is important to understand the mechanism to extract sufficient performance
Basic Concepts, First of All

• Virtual ..., Logical ...
  – Virtual addresses, logical devices, logical sectors, virtual machines
  – To handle as if it is ...

• Real ..., Physical ...
  – Real addresses, physical devices, phisical sectors
  – Itself, as it is
Virtualization: As if … but …

- As if it is large, but it actually small
- As if it is flat, but it actually uneven
- As if there are many, but there is actually one
- As if exclusively usable, actually shared

Virtualization is magic to hide complexity or individual dependency; as it is magic, there is a trick = Mapping between the real and the virtual Translating so that it looks as if …
Cost of Virtualization

- We do virtualize as its merits are greater than its demerits, but virtualization does not always mean positive results.
Physical Memory and Virtual Memory

- Most of ordinary embedded device projects so far have handled only physical memory.
- Recently, as the size of embedded systems grow, PC-oriented OSes such as Linux and WindowsCE are getting widely used; these operating systems provide virtual memory systems.
- This presentation explains Linux.
Physical Memory

- Single memory space
- As each device is implemented with different addresses for ROM, RAM and I/O, programmers should code accordingly
Virtual Memory

• Merits
  – User programs do not depend on actual memory map (implementation address, implementation size) any more
  – Can use non-contiguous physical memory fragments as contiguous virtual memory
  – Memory protection: Can prevent irrelevant memory from being destroyed by bugs

• Introducing new concepts
  – Address translation
  – Multiple memory spaces
  – Demand paging
Conceptual Schema of Virtual Memory

Virtual Address Space

Physical Memory Space

Swapping Area

Address Translation

Virtual Add Space

address x

MMU

Linux kernel controls

Physical Addr Space

ROM

address y

RAM

I/O

I/O

I/O
Virtual Memory is only in CPU

Only physical addresses come out of CPU onto address bus. Virtual addresses cannot be observed with the logic analyzer.
User Program Handles only Virtual Addresses

Physical addresses are handled only in kernel mode, i.e. kernel itself and device drivers
Address Translation with MMU

Virtual Addr

Page Directory
Base Register

Directory Index
Page Table Index
Page Offset

Page Directory

Directory Entry

Page Table

Page Table Entry

Physical Addr

Page

No page! ... page fault

Virtual Addr
**TLB**

<table>
<thead>
<tr>
<th>Virtual Address Pages</th>
<th>Physical Address Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Translation Lookaside Buffers
- Something like hashtable getting a physical address by a using virtual address as a key
- In most address translation, page is found in TLB, so there is no need to access page directory or page table
Multiple Memory Spaces

Independent virtual memory spaces per process

Physical Addr Space

ROM

RAM

I/O

I/O
Demand Paging

• Mapped per page
  – Page size is usually 4Kbytes

• Two phase execution
  1. Virtual memory is allocated (mmap); just registered in management table
  2. At actual access, physical memory is allocated for the page

As no physical page is allocated unless the page is accessed

Virtual memory size \geq \text{Actually required physical memory size}
Example of Demand Paging Behavior

(1)

Page fault occurs; Transits into kernel mode

Virtual Addr Space  Physical Addr Space

read access

No Corresponding Physical page!
Example of Demand Paging Behavior (cont.)

Kernel loads the data and maps physical address

Virtual Addr Space  Physical Addr Space

Mapping

DMA Transfer
Example of Demand Paging Behavior (cont.)

(3)

Virtual Addr. Space                  Physical Addr Space

data

Return to user mode;
User program can read the data as if nothing happened
(but time has elapsed actually)
Data read from disk are kept on memory as far as space allows. Access tends to be sequential, so several pages are read at a time in advance; Thus disk access does not occur every time in (2)
If no physical memory is available in (2), a page assumed to be least used is released. If the contents of this page is not modified, it is just discarded; otherwise, the page is swapped out onto swap device.

Many of embedded Linux does not have a swap device.
A requested page is allocated using area of a page released. This “juggling” enables to larger size of virtual memory than physical memory size actually installed.
The same pages of a same file will be shared among more than one processes; for both read-only pages and writable pages.
Copy on Write

If write operation occurs on writable and private page...

write access

r/w private

Page Fault Occurs

read only
Copy on Write (cont.)

Process A

write access

Process B

Kernel copies the page and changes the page status to read/write
Memory Spaces of Processes

- About 3 GB user memory space per process
- Kernel space is shared among processes; kernel space is not allowed to read/write/execute in user mode;
- User memory spaces are switched when processes switched

**TASK_SIZE**
- 0x00000000 for i386
- 0xffffffff
  - TASK_SIZE is 0xc0000000 for i386; 0xbf000000 for ARM
Example of Memory Space of a User Process

cat /proc/<PROCESS_ID>/maps

<table>
<thead>
<tr>
<th>Address Range</th>
<th>file offset</th>
<th>inode</th>
<th>file name</th>
</tr>
</thead>
<tbody>
<tr>
<td>00101000-0011a000</td>
<td>00000000</td>
<td>fd:00</td>
<td>15172739 /lib/ld-2.4.so</td>
</tr>
<tr>
<td>0011a000-0011b000</td>
<td>00018000</td>
<td>fd:00</td>
<td>15172739 /lib/ld-2.4.so</td>
</tr>
<tr>
<td>0011b000-0011c000</td>
<td>00019000</td>
<td>fd:00</td>
<td>15172739 /lib/ld-2.4.so</td>
</tr>
<tr>
<td>0011e000-0024a000</td>
<td>00000000</td>
<td>fd:00</td>
<td>15172740 /lib/libc-2.4.so</td>
</tr>
<tr>
<td>0024a000-0024d000</td>
<td>0012b000</td>
<td>fd:00</td>
<td>15172740 /lib/libc-2.4.so</td>
</tr>
<tr>
<td>0024d000-0024e000</td>
<td>0012e000</td>
<td>fd:00</td>
<td>15172740 /lib/libc-2.4.so</td>
</tr>
<tr>
<td>0024e000-00251000</td>
<td>0024e000</td>
<td>00:00</td>
<td>0</td>
</tr>
<tr>
<td>08048000-08049000</td>
<td>00000000</td>
<td>fd:00</td>
<td>11666681 /home/koba/lab/loop/a.out</td>
</tr>
<tr>
<td>08049000-0804a000</td>
<td>00000000</td>
<td>fd:00</td>
<td>11666681 /home/koba/lab/loop/a.out</td>
</tr>
<tr>
<td>b7fef000-b7ff1000</td>
<td>b7fef000</td>
<td>00:00</td>
<td>0</td>
</tr>
<tr>
<td>b7fff000-b8000000</td>
<td>b7fff000</td>
<td>00:00</td>
<td>0</td>
</tr>
<tr>
<td>bffeb000-c0000000</td>
<td>bffeb000</td>
<td>00:00</td>
<td>0 [stack]</td>
</tr>
</tbody>
</table>

```
Address Range  file offset       inode     file name
r: read         device
w: write        major:minor
x: execute      s: shared
p: private (copy on write)
```
Example of Memory Space of a User Process (Detail)

cat /proc/<PROCESS_ID>/smaps

....
0011e000-0024a000 r-xp 00000000 fd:00 15172740  /lib/libc-2.4.so
Size: 1200 kB
Rss: 136 kB
Shared_Clean: 136 kB
Shared_Dirty: 0 kB
Private_Clean: 0 kB
Private_Dirty: 0 kB
0024a000-0024d000 r-xp 0012b000 fd:00 15172740  /lib/libc-2.4.so
Size: 12 kB
Rss: 8 kB
Shared_Clean: 0 kB
Shared_Dirty: 0 kB
Private_Clean: 0 kB
Private_Dirty: 8 kB
0024d000-0024e000 rwxp 0012e000 fd:00 15172740  /lib/libc-2.4.so
Size: 4 kB
Rss: 4 kB
Shared_Clean: 0 kB
Shared_Dirty: 0 kB
Private_Clean: 0 kB
Private_Dirty: 4 kB
....

RSS = Physical Memory Size
mmap System Call

```
#include <sys/mman.h>

void *mmap(void *start, size_t length, int prot, int flags,
           int fd, off_t offset);

int munmap(void *start, siz_t_t length);
```

- Map/Unmap files or devices onto memory
- Argument `prot`
  - PROT_NONE, or OR operation of PROT_EXEC, PROT_READ, and PROT_WRITE
- Argument `flags`
  - MAP_FIXED, MAP_SHARED, MAP_PRIVATE, MAP_ANONYMOUS, ...
mmap tips

- Unless specified as MAP_FIXED, kernel searches available pages
- If MAP_FIXED is specified and it overlaps existing pages, the pages are *mumpapped* internally
  - Thus this option is usually not used
- File offset must be multiple of page size
- Addresses and sizes of *mmap* and *munmap* need not be identical
Usage of mmap (1)

• As substitute of malloc for large size
  – No data copy, such as compaction, occurs
  – Unlike malloc/free, addr and size at munmap can be different than those at mmap
    • It is possible to allocate a large chunk with a single mmap, and to release piecemeal with multiple munmaps
  – In malloc of glibc implementation, mmap is called for a certain size or larger
    • DEFAULT_MMAP_THRESHOLD = (128*1024)
Usage of mmap (2)

• Fast file access
  – In system calls read and write, data is internally buffered in physical pages; from there data is copied to array specified by user
  – Using mmap enables to access page directly, thus number data copies can be reduced
  – java.nio.MappedByteBuffer in Java 1.4
Usage of mmap (3)

• Shared memory among processes
  – Map the same file as readable/writable and shared from more than one processes
  – IPC shared memory system calls (*shmget*, *shmat*, ...) does above internally
Usage of mmap (4)

• Access to physical memory, I/O ports
  – By mapping device file /dev/mem, it becomes possible to read/write physical memory space in user mode
  – To access /dev/mem, root privilege is required
Summary

• Virtual memory usage and physical memory usage are not same. Physical one matters in practice
• Be careful when overhead of virtual memory occurs.
  – TLB miss
  – Page fault
• Make use of system call mmap
References

• Linux kernel source code
  http://www.kernel.org/

• GNU C library source code
  http://www.gnu.org/software/libc/

• “Understanding the Linux Kernel (2nd Edition)”
  by Daniel P. Bovet (O’Reilly) [Japanese translation; 3rd Edition available in English]

• “Linux kernel 2.6 Kaidokushitsu”, by Hirokazu Takahashi et. al. (SoftBank Creative) [in Japanese]

• Linux man command

• And other search results on web
One more thing: hot topics

• From CELF BootTimeResources
  – KernelXIP
  – ApplicationXIP
  – (DataReadInPlace)
• From CELF MemoryManagementResources
  – Huge/large/superpages
  – Page cache compression