Predictive bitmaps

An experiment in speeding up Linux demand paging

Andi Kleen

Caveat

Experiment

Work in progress

Historical background

- Old home computer OS didn't have VM
  - Loading ...
  - AmigaOS, C64
- Original Unix written for 16bit PDP/11
  - Each process had up to three 16bit segments
  - Processes were swapped in/out of memory completely
- VM on VAX in the 80ies
  - 2-4MB memory
  - 2K pages
  - Demand Paging on BSD
- Pages are read/created only on first touch
  - Complete process swapping still possible

Demand paging

- Linux fully demand paged
  - Just VMAs and file handles exist initially
  - Even page tables are only created on demand
- Naive view demand paging gives optimal memory use
  - More difficult to age pages only needed once
- Access patterns scattered
  - Depending on how the program is executed
  - Bad for IO subsystem
- Everything is done one page at a time
  - There is readahead, but it often doesn’t help
  - Not much batching possible

Pages

- On x86 pages normally 4K
  - 8K, 16K, 64K also on other architectures
  - Very small for modern memory sizes
  - large pages optional, handled outside normal VM in hugetlbfs
- Kernel keeps a page cache
  - Pages can be mapped into process address space
  - Cached in the background
    - write/read just copy to/from them
    - mmap (including executable mappings) access directly
  - Also used for metadata by many file systems
  - All the same radix tree data structure internally

mmaped IO

Used for executables, shared libraries, other files (with mmap)

- First just virtual address space is reserved
- On fault the kernel creates page table and maps the page in
  - Searches the page in the page cache
  - When cached just create reference ("minor fault")
  - When uncached read from disk ("major fault")
  - Even minor fault relatively expensive in CPU time
    - Tens of thousands of cycles
- Sophisticated readahead algorithms in page cache
  - Detects sequential patterns
  - Initiates readahead in the background
  - Readahead window is grown automatically
  - Special case for full file access
  - Limit of pinned pages (2MB by default)

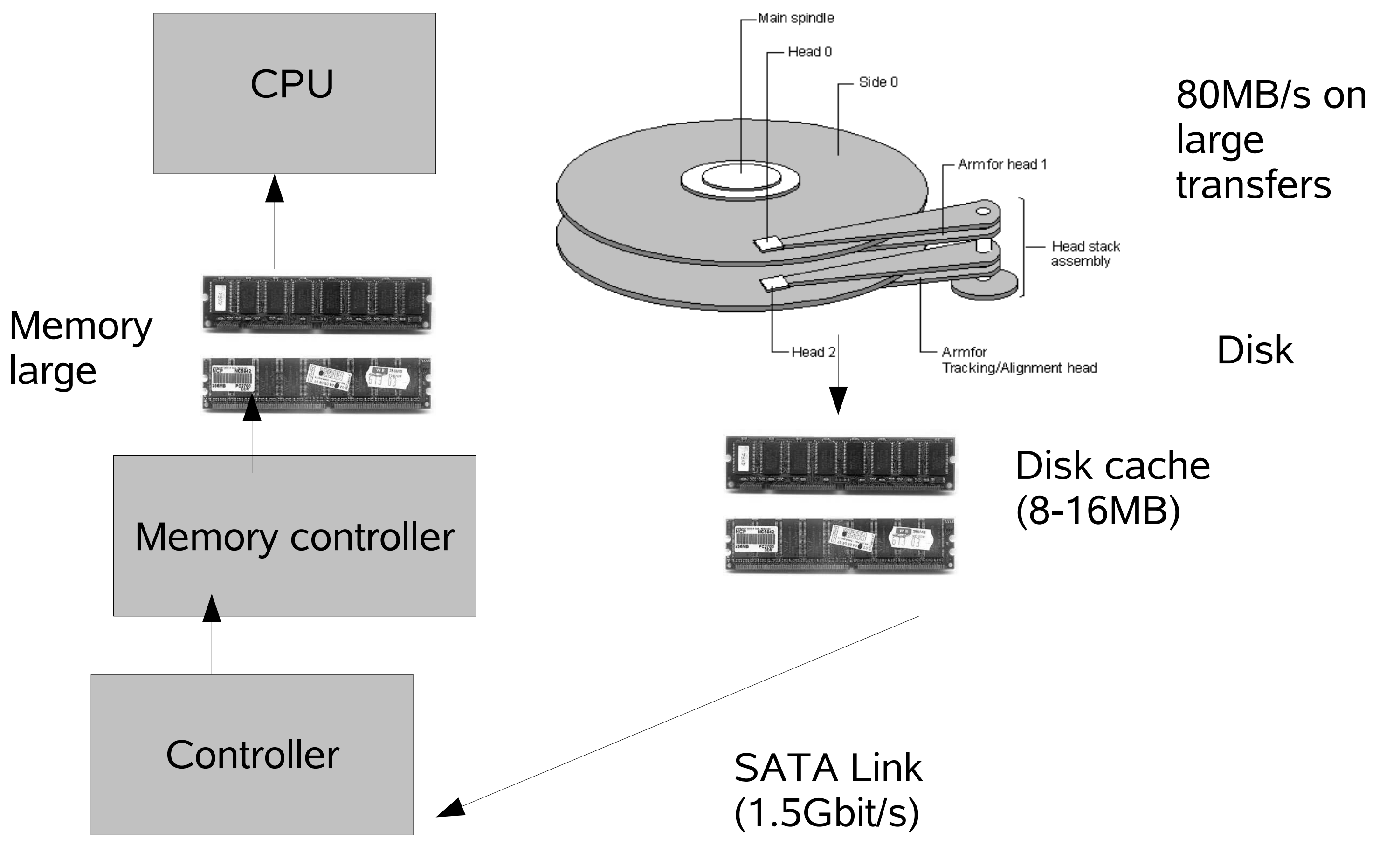
Sample address space layout

```
# cat /proc/self/maps
00400000-00405000 r-xp 00000000 08:03 973466          /bin/cat
00604000-00606000 rw-p 00004000 08:03 973466          /bin/cat
01c6e000-01c8f000 rw-p 01c6e000 00:00 0               [heap]
7f89908d8000-7f8990a14000 r-xp 00000000 08:03 1460168     /lib64/libc-2.6.1.so
7f8990a14000-7f8990c13000 ---p 0013c000 08:03 1460168     /lib64/libc-2.6.1.so
7f8990c13000-7f8990c16000 r--p 0013b000 08:03 1460168     /lib64/libc-2.6.1.so
7f8990c16000-7f8990c18000 rw-p 0013e000 08:03 1460168     /lib64/libc-2.6.1.so
7f8990c18000-7f8990c1d000 rw-p 7f8990c18000 00:00 0
7f8990c1d000-7f8990c39000 r-xp 00000000 08:03 1460357     /lib64/ld-2.6.1.so
7f8990cee000-7f8990d2d000 r--p 00000000 08:03 324805      /usr/lib/locale/en_US.utf8/LC_CTYPE
[...]
7f8990e35000-7f8990e36000 r--p 00000000 08:03 308435      /usr/lib/locale/en_US.utf8/LC_IDENTIFICATION
7f8990e36000-7f8990e38000 rw-p 7f8990e36000 00:00 0
7f8990e38000-7f8990e3a000 rw-p 0001b000 08:03 1460357     /lib64/ld-2.6.1.so
```

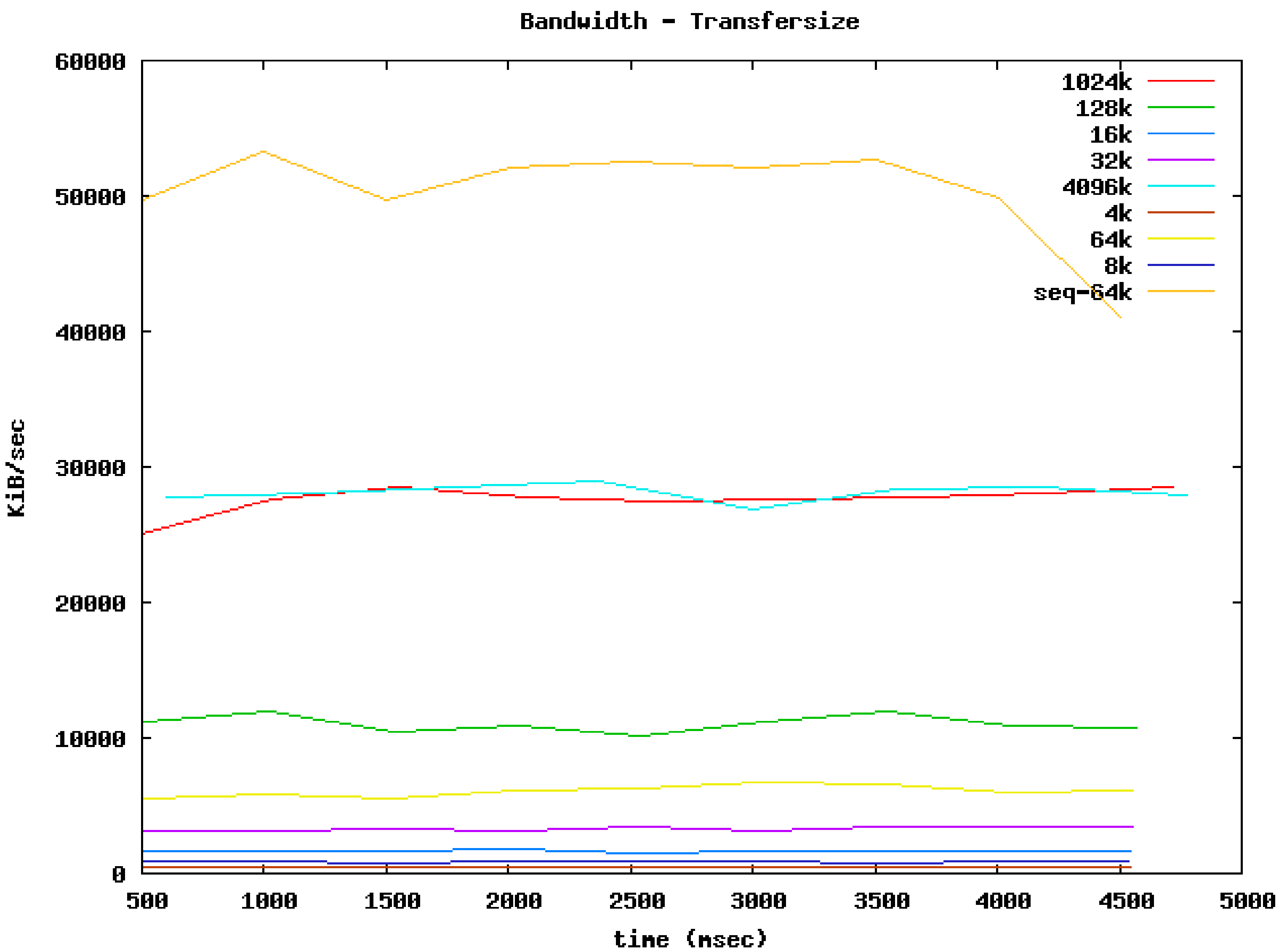
Trends

- Memory is getting faster and bigger
  - But slower than CPUs.
  - Latency doesn't improve that much
- Hard disks are also getting faster and bigger
  - But much slower than memory again
  - Latency is still quite poor
- Swapping is slow
  - Especially swapping in
  - Seeks cost

Disk system overview



Disk transfer rate



Hard disk facts

- Hard disk reasonably fast
  - 80+MB/s on a 7200rpm disk with little seeking
  - For sequential transfers
- On seeky workloads transfer rate drops dramatically
  - Depends on spin rate
  - But even on fast disks it drops <10MB/s
- Connection to hard disks (SATA) has plenty of bandwidth
  - And hard disk has a large buffer (8MB)
- Each seek+transfer reads a complete track into disk memory
  - Getting whole track (nearly) free
  - Assuming file is not fragmented
- Unit of useful IO
  - Currently around 1MB.
  - Growing in each generation

Exception: Solid State Device

- Flash disks becoming larger and cheaper
  - Expect they will replace HDs at some future point
  - Currently still far more expensive and smaller
- Eliminate seek time (mostly)
  - Still some command overhead
  - But most of them are currently slow for writing
- Unit of useful IO is still quite large
  - Due to erase block sizes on writes
  - And requirement to use multiple chips in parallel
- But still HDs will be with us for a long time
  - And best flash tuning still under study

Disclaimer

This presentation does not cover flash

How a program is loaded

- ELF loader reads the header / PHDRs (few KB only)

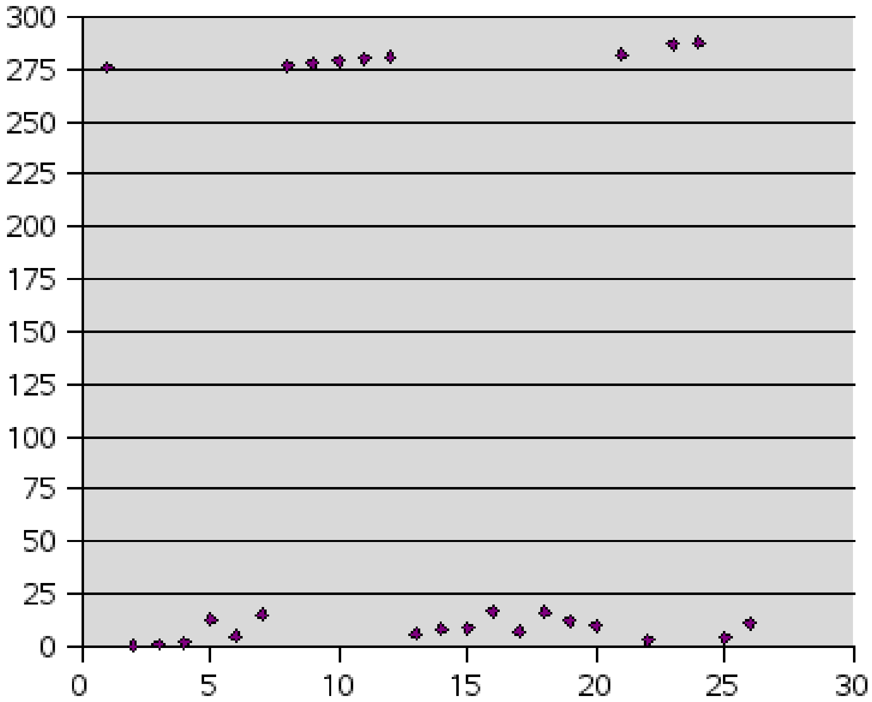
- Maps all LOAD segments in
  - Not even page tables are setup
  - But doesn’t do any IO
- Starts "interpreter" (ld.so)
  - Reads headers of the shared libraries
  - Maps them in
  - IO only on headers
- Program starts executing
  - Each page fault on .text reads a 4K block from disk
  - A lot of seeks typically

Executable access pattern

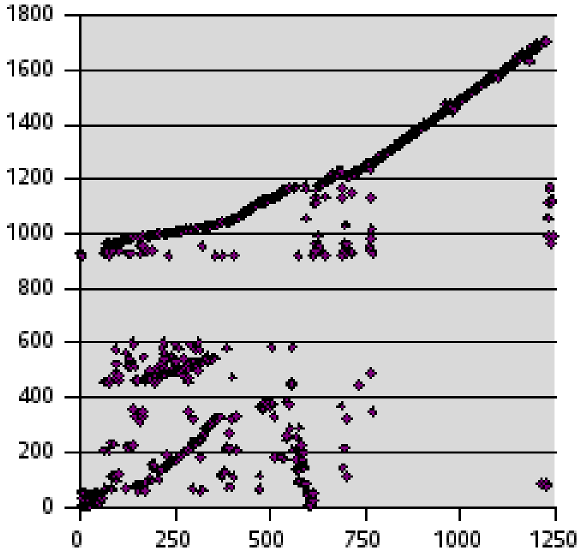
/bin/ls using a systemtap script

```
# cat fault.stp
probe vm.pagefault  {
        if (task_execname(task_current()) == "ls" && address < 0x100000000) {
                printf("%u\n", address);
        }
}
# stap fault.stp
# ls
```

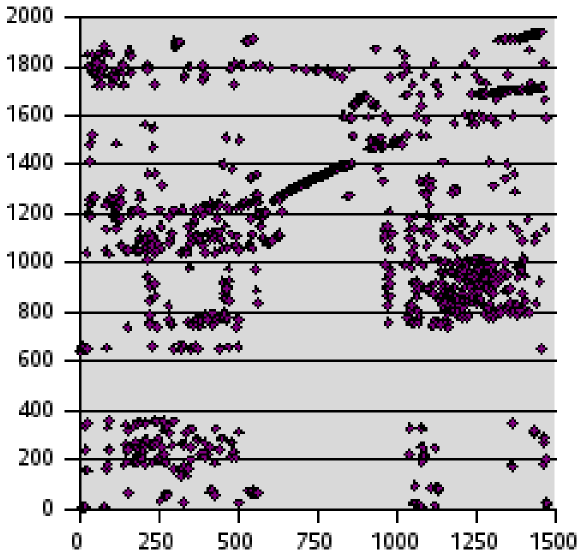
ls faults



gdb /bin/ls + run faults



emacs faults



What happens when we disable demand paging?

- Simple patch in kernel
  - echo 1 > /proc/sys/vm/mmap_slurp_all
- Always read mapping completely during mmap
  - Can be done asynchronously
  - Right now synchronously
- Disadvantage
  - More memory use
  - More IO
  - Doesn’t interact well with mmap for normal file IO

mmap_slurp_all results

No GUI programs because they’re difficult to time

- All with cold caches (echo 1 > /proc/sys/vm/drop_caches)
- gcc -O2 -S sched.i
  - CPU: 1.57[sd 0.05] -> 1.42[sd 0.02] -10%
  - MEM: 5787MB[sd 143] -> 6968MB [sd 130] +17%
- emacs -batch
  - CPU: 0.85s[sd 0.05] -> 0.59s[sd 0.05] -31%
  - MEM: 10834MB[sd 181] -> 11968B[sd 199] +10%
- gdb -batch
  - CPU: 0.18s[sd 0.02] -> 0.12s[sd 0]
  - MEM: 2836MB[sd 130] -> 4274MB[sd 86]

pbitmaps

- Idea originally from DG/UX
- echo 1 > /proc/sys/vm/pbitmap_enabled
- Add a new ELF section on disk that contains one bit for each program page
  - (except anonymous which is not known in advance)
- Bitmap updated on program exit based on the page tables
- On exec prefault all the pages in the bitmap
  - Also precreate BSS pages

Implementation

- Based on sys_readahead
  - Uses some throttling based on free memory on local node
  - Ignores block congestion currently
- Synchronous
  - Could use a prefetch thread for larger working sets
- Three phases
  - Read bitmap
  - Start readahead on bitmap
  - Fault pages in
- PHDR vs SHDR
  - PHDR would be better but requires relinking
  - SHDR is additional seek, but can be added easily
  - Using SHDR hack right now

Access rights

- O_FORCEWRITE for bitmap write to mapped executable
  - Normally forbidden
  - No security implications
  - Flag currently exposed to user space
- Executables have to writable by executing uid
  - Without it just the last prefetch state is used
- Could force write in kernel
  - New in-kernel credentials infrastructure from AFS
  - Or just changing uid temporarily
  - Need to think through the security implications
  - Problem are advanced security models like selinux, AA, smack

pbitmap I

- Optional batching of page faults (early_fault sysctl)
  - Makes IO synchronous at read time
  - Better use of CPU cache
  - Could be even more optimized to batch locks etc.
- Causes a write to disk
  - Similar to infamous atime update
  - Only done each interval (60s) to avoid thrashing
  - Could also compare bitmaps (not done currently)
  - Executables have to be writable to user currently

pbitmap results

with early_fault

- gcc -O2 -S sched.i
  - CPU: 1.57s[sd 0.05] -> 1.42s[sd 0.02] -> 1.46s[sd 0.02] -8%
  - MEM: 5787MB[sd 143] -> 6968MB [sd 130] -> 6000MB[sd 87] +3.5%
- emacs -batch
  - CPU: 0.85s[sd 0.05] -> 0.59s[sd 0.05] -> 0.83s[sd 0.01] -3.3%
  - MEM: 10834MB[sd 181] -> 11968B[sd 199] -> 10875MB [sd 142] +1%
- gdb -batch
  - CPU: 0.18s[sd 0.02] -> 0.12s[sd 0] -> 0.17s [sd 0.01] -0.05%
  - MEM: 2836MB[sd 130] -> 4274MB[sd 86] -> 2900MB [sd 181] +2.2%

Result

Experimental results did not help that much

Improvements from simple pbitmap code only a few percent

mmap_slurp helps more, but it has other drawbacks

Why did pbitmap not help as much as expected?

Some speculation

- Readahead algorithms are already pretty good
  - Readahead code will already do large IOs after window ramped up
- early_fault is likely a bad idea
  - Adds too much waiting for IO
  - Cache effects not worth it
- Too synchronous
  - Complete prefetch procedure should be background
- Pages accumulate over time currently
  - Because the previous run pages always get faulted in too
  - And the bitmap write at the end doesn’t know
  - Need an aging mechanism

pbitmap other issues

- Adding new header can break installation disk layout
  - But currently preallocation leaves holes on installs
  - pbitmap.c only appends/rewrites
- Doesn’t handle shared libraries right now
  - Would need ld.so support and a new syscall
  - Or just use mmap_slurp_all
- Executable changes
  - rpm -V breaks
  - Can be handled similar to prelink using rpm scripts
  - May cause larger incremental backups etc.
- Include bitmaps by default in executables?
  - Also might need ELF official section numbers?

Conclusion

Wake up! presentation is over.

- Should do swapping and demand paging for .text in larger chunks
  - Best way to do that still under research
- pbitmaps interesting, but first implementation not full success
  - Needs more work
- ftp://ftp.firstfloor.org/pub/ak/pbitmap/ (in quilt format)
  - pbitmap.c (to add pbitmap SHDR)
- Questions: andi@firstfloor.org

Backup

Future improvements

- Do pure background prefetch without early_fault
- Do more instrumentation where time is spent
  - Using seekwatch, blktrace, more systemtap
  - Tune prefetch distances
  - Tune interaction with standard page cache prefetch
- Do it in user space with "prefetch server"?
  - One of the review comments
  - Couldn’t force write there

Program headers

```
# readelf -l  /bin/ls

Elf file type is EXEC (Executable file)
Entry point 0x402410
There are 10 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x0000000000000230 0x0000000000000230  R E    8
  INTERP         0x0000000000000270 0x0000000000400270 0x0000000000400270
                 0x000000000000001c 0x000000000000001c  R      1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x00000000000150f4 0x00000000000150f4  R E    200000
  LOAD           0x0000000000015d50 0x0000000000615d50 0x0000000000615d50
                 0x0000000000000740 0x0000000000000ce0  RW     200000
  DYNAMIC        0x0000000000015de0 0x0000000000615de0 0x0000000000615de0
                 0x00000000000001c0 0x00000000000001c0  RW     8
  NOTE           0x000000000000028c 0x000000000040028c 0x000000000040028c
                 0x0000000000000020 0x0000000000000020  R      4
  NOTE           0x00000000000002ac 0x00000000004002ac 0x00000000004002ac
                 0x0000000000000018 0x0000000000000018  R      4
  GNU_EH_FRAME   0x0000000000013330 0x0000000000413330 0x0000000000413330
                 0x000000000000061c 0x000000000000061c  R      4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     8
  GNU_RELRO      0x0000000000015d50 0x0000000000615d50 0x0000000000615d50
                 0x0000000000000280 0x0000000000000278  R      1
```

ELF file II

section headers

```
# readelf -S /bin/ls
There are 31 section headers, starting at offset 0x16a00:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000400270  00000270
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.ABI-tag     NOTE             000000000040028c  0000028c
       0000000000000020  0000000000000000   A       0     0     4
  [ 3] .note.SuSE        NOTE             00000000004002ac  000002ac
       0000000000000018  0000000000000000   A       0     0     4
  [ 4] .hash             HASH             00000000004002c8  000002c8
       0000000000000320  0000000000000004   A       6     0     8
  [ 5] .gnu.hash         GNU_HASH         00000000004005e8  000005e8
       0000000000000060  0000000000000000   A       6     0     8
  [ 6] .dynsym           DYNSYM           0000000000400648  00000648
       0000000000000978  0000000000000018   A       7     1     8
  [ 7] .dynstr           STRTAB           0000000000400fc0  00000fc0
       000000000000045d  0000000000000000   A       0     0     1
  [ 8] .gnu.version      VERSYM           000000000040141e  0000141e
       00000000000000ca  0000000000000002   A       6     0     2
...
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
```