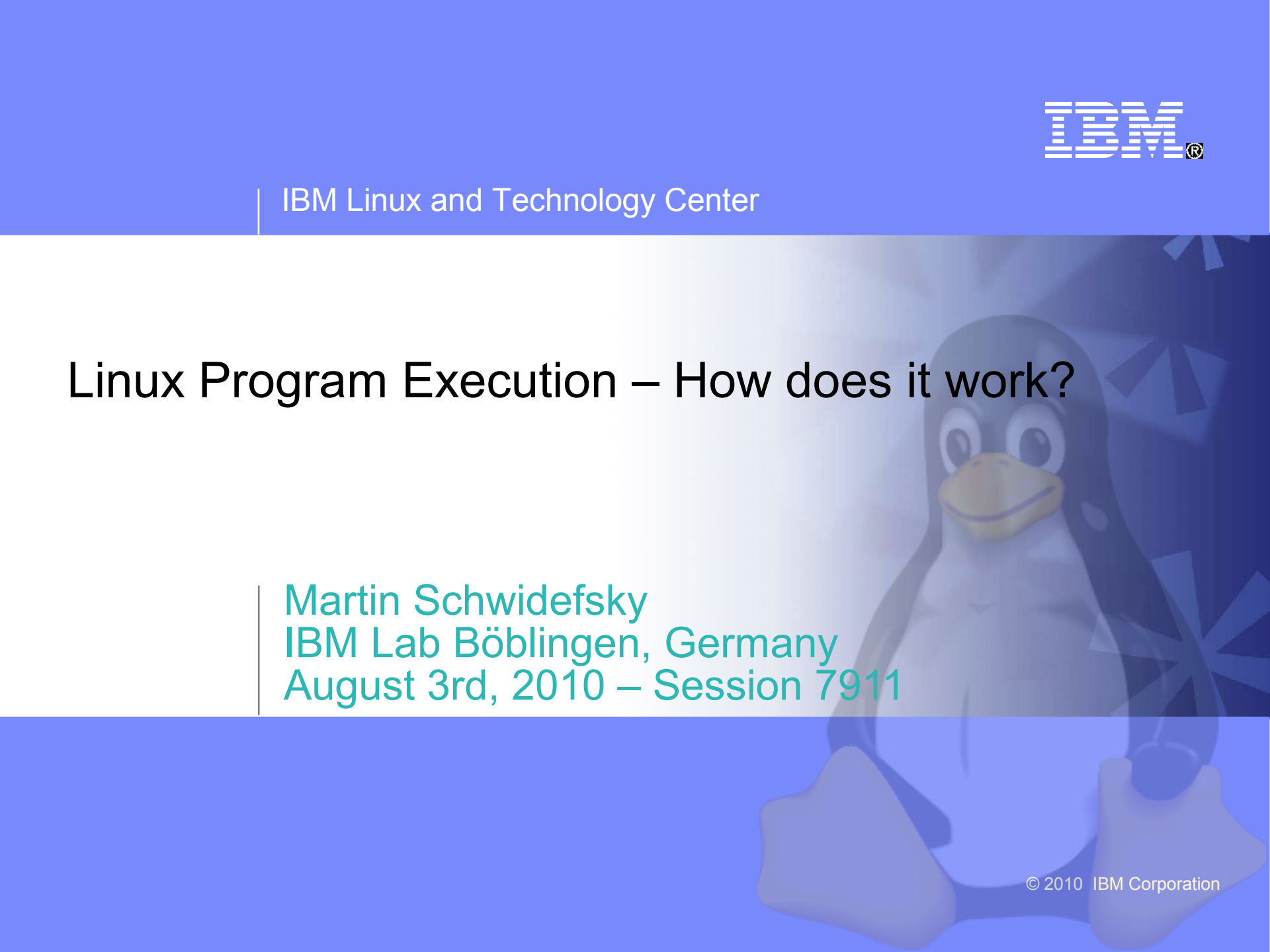


Linux Program Execution – How does it work?

Martin Schwidefsky
IBM Lab Böblingen, Germany
August 3rd, 2010 – Session 7911

Trademarks & Disclaimer

The following are trademarks of the International Business Machines Corporation in the United States and/or other countries. For a complete list of IBM Trademarks, see www.ibm.com/legal/copytrade.shtml:

IBM, the IBM logo, System p, System Storage, System x , and System z are trademarks of IBM Corporation in the United States and/or other countries. For a list of additional IBM trademarks, please see http://ibm.com/legal/copytrade.shtml.

The following are trademarks or registered trademarks of other companies: Java and all Java based trademarks and logos are trademarks of Sun Microsystems, Inc., in the United States and other countries or both Microsoft, Windows,Windows NT and the Windows logo are registered trademarks of Microsoft Corporation in the United States, other countries, or both. Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.
UNIX is a registered trademark of The Open Group in the United States and other countries or both. Linux is a trademark of Linus Torvalds in the United States, other countries, or both. Cell Broadband Engine is a trademark of Sony Computer Entertainment Inc. InfiniBand is a trademark of the InfiniBand Trade Association.
Other company, product, or service names may be trademarks or service marks of others.

NOTES: Linux penguin image courtesy of Larry Ewing (lewing@isc.tamu.edu) and The GIMP

Any performance data contained in this document was determined in a controlled environment. Actual results may vary significantly and are dependent on many factors including system hardware configuration and software design and configuration. Some measurements quoted in this document may have been made on development-level systems. There is no guarantee these measurements will be the same on generally-available systems. Users of this document should verify the applicable data for their specific environment. IBM hardware products are manufactured from new parts, or new and serviceable used parts. Regardless, our warranty terms apply.
Information is provided “AS IS” without warranty of any kind. All customer examples cited or described in this presentation are presented as illustrations of the manner in which some customers have used IBM products and the results they may have achieved. Actual environmental costs and performance characteristics will vary depending on individual customer configurations and conditions.

This publication was produced in the United States. IBM may not offer the products, services or features discussed in this document in other countries, and the information may be subject to change without notice. Consult your local IBM business contact for information on the product or services available in your area.
All statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only.
Information about non-IBM products is obtained from the manufacturers of those products or their published announcements. IBM has not tested those products and cannot confirm the performance, compatibility, or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Prices are suggested US list prices and are subject to change without notice. Starting price may not include a hard drive, operating system or other features. Contact your IBM representative or Business Partner for the most current pricing in your geography. Any proposed use of claims in this presentation outside of the United States must be reviewed by local IBM country counsel prior to such use. The information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any

Example program

- The goal of this presentation is to give an idea what happens in Linux to execute this simple “Hello World” program:

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
        char *p = malloc(1024);
        printf("Hello world %p\n", p);
        return 0;
}
```

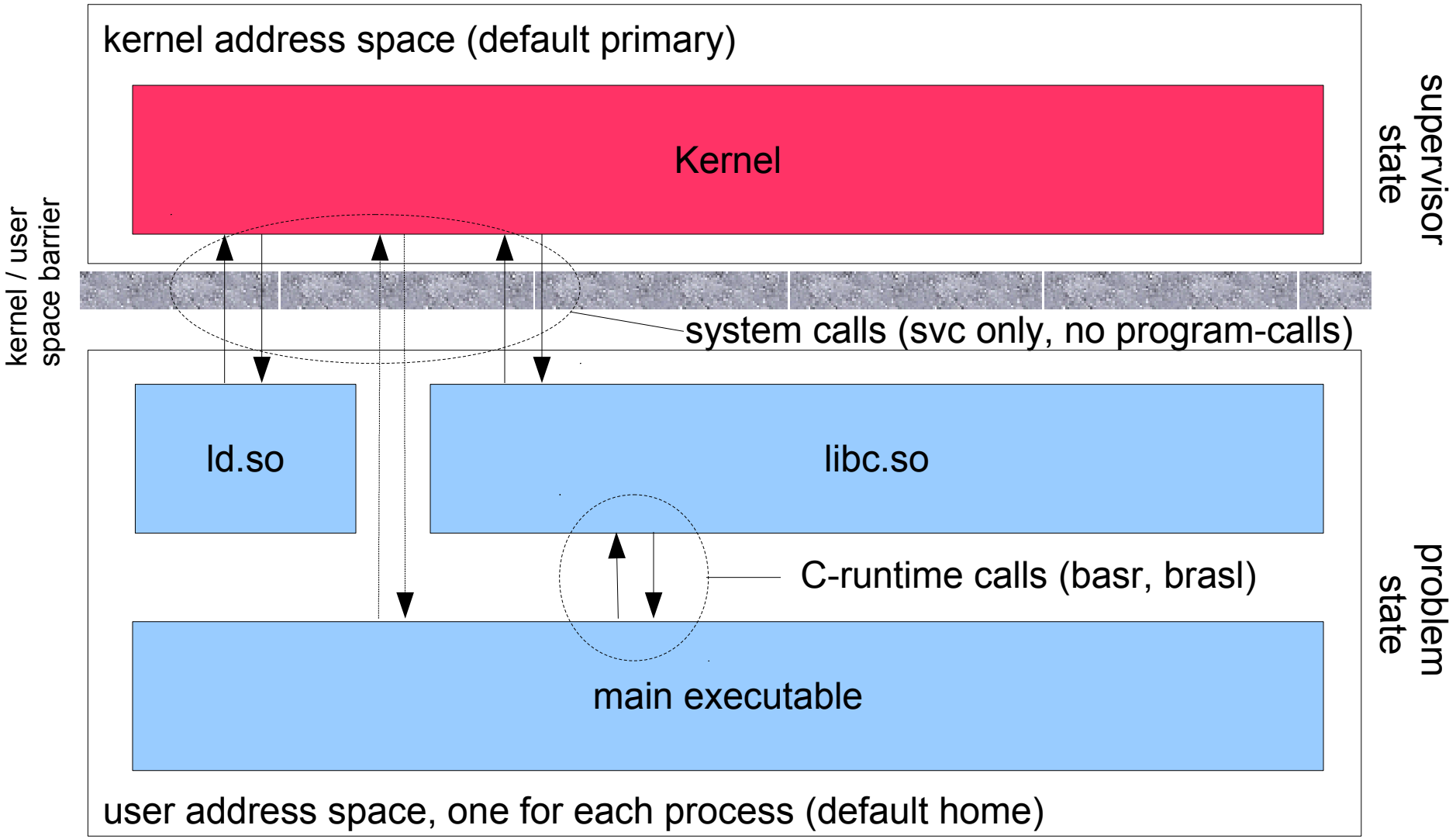
Unix design points

- Portable, multi-tasking and multi-user operating system
- Plain text for storing data
- “Everything is a stream of bytes”
- A hierarchical file system
  - Arbitrarily nested subdirectories
  - Devices are represented as special files
- Use of small programs that are connected through a command line interpreter and pipes
- Heavy use of the C programming language
- Division between user-space and kernel-space
- POSIX standard
  - Portable Operating System Interface [for Unix]

POSIX.1 – Core Services

- Process Creation and Control
  - Executable machine code (kernel threads vs. user space threads)
  - Virtual memory with code, data, a call stack
  - Resource descriptors allocated to the process
  - Security attributes: process owner, permissions
  - Processor state (context)
- Signals, Timers
  - Notifications sent to a process via an asynchronous function call
  - Timing events delivered as signals
- File and Directory Operations
- Pipes
- .. and some more

Simplistic system layout

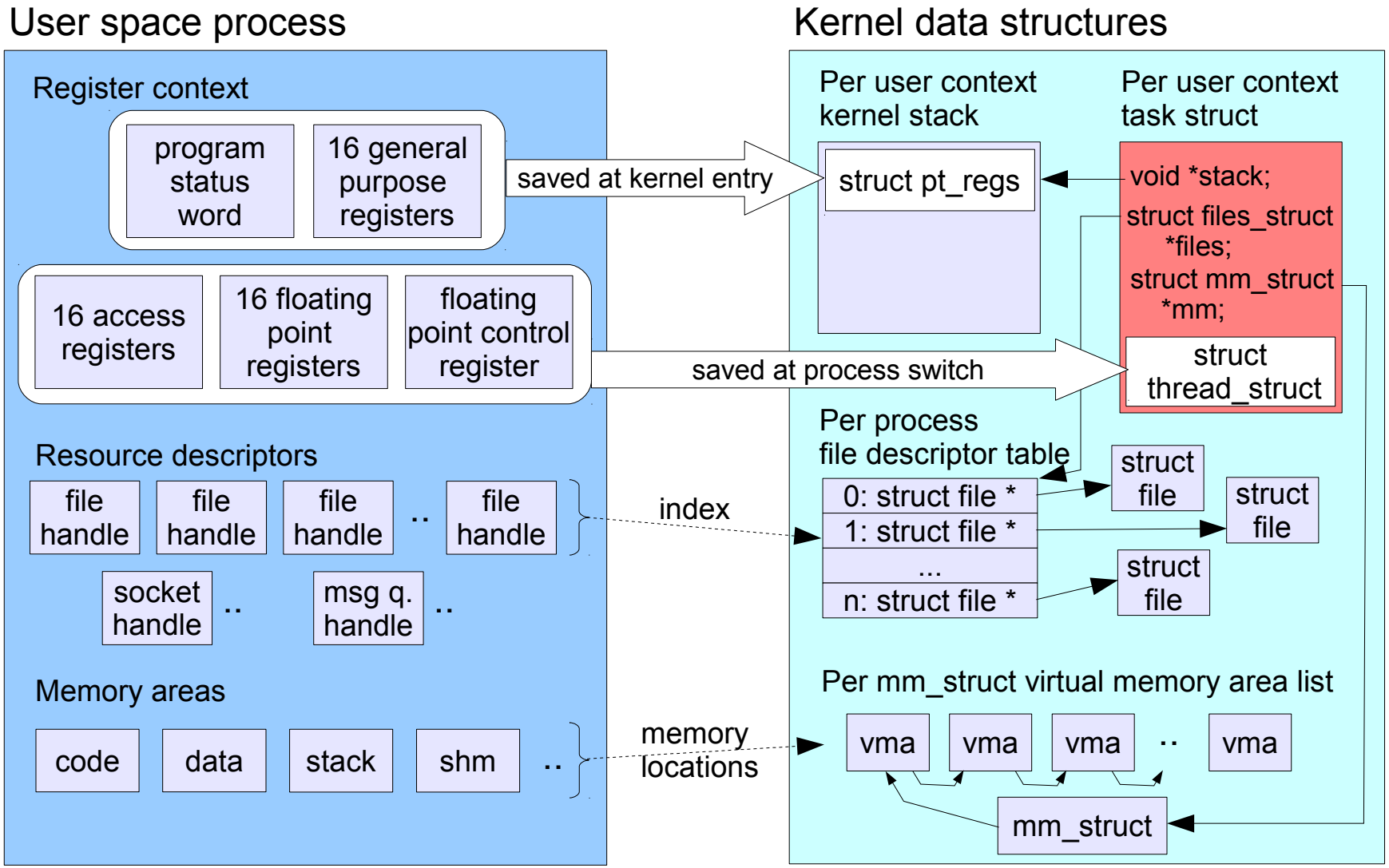


Kernel access to user data (uaccess)

- Strict separation between the kernel and user address space
  - Kernel address space uses a three level page table, max 4TB
  - User address space uses two, three or four level page table, max 8PT
- **No kernel data structures visible in user space**
  - In particular the lowcore pages are not mapped to user space
  - Usually there is nothing mapped at address zero
- The uaccess functions provide access to user data for kernel code

	copy to user	copy from user	futex
kernel in primary machine < z9	mvcs	mvcp	sacf to secondary + compare-and-swap
kernel in primary machine >= z9	mvcos	mvcos	sacf to secondary + compare-and-swap
kernel in home machine < z9	page table walk	page table walk	page table walk + compare-and-swap
kernel in home machine >= z9	mvcos	mvcos	page table walk + compare-and-swap

User space context & kernel data structures



Frequently used system calls

- Working with files
  - open, dup, read, write, lseek, close, stat, creat, truncate, ..
- Filesystem operations
  - link, unlink, rename, mkdir, rmdir, ..
- Virtual memory operations
  - brk, mmap, munmap, mprotect, mlock, munlock
- Process create, execution, termination
  - fork, clone, execve, exit, pause, wait4, pipe, chdir, ..
- Signals
  - kill, signal, sigaction, sigprocmask, sigreturn, ..
- .. and many more, currently there are 332 system calls for Linux on System z, some of them multiplexer e.g. ipc

strace of statically linked “Hello World”

```
> cat hello.c
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
      char *p = malloc(1024);
      printf("Hello world %p\n", p);
      return 0;
}

> gcc -static -o hello -O2 hello.c

> strace -v ./hello
execve("./hello", ["./hello"], ["HOSTNAME=t6360015", "TERM=xterm",
       "SHELL=/bin/bash", "HISTSIZE=1000",
       "SSH_CLIENT=9.152.212.37 56750 22"...,
       "SSH_TTY=/dev/pts/0", "USER=root", "USERNAME=root",
       "PATH=/sbin:/bin:/usr/sbin:/bin"..., "PWD=/root", "SHLVL=1",
       "HOME=/root", "BASH_ENV=/root/.bashrc", "LOGNAME=root",
       "_=/usr/bin/strace"]) = 0
brk(0) = 0x8009d000
brk(0x8009df60) = 0x8009df60
brk(0x800bef60) = 0x800bef60
brk(0x800bf000) = 0x800bf000
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
     = 0x20000000000
write(1, "Hello world 0x8009e5e0\n"..., 23) = 23
exit_group(0) = ?
```

Binary formats

- Linux knows about different binary formats
  - aout: the oldest executable format for Linux, does not exists for System z
  - elf: Executable and Linkage Format, the standard executable format
  - em86: wrapper for /usr/bin/em86 to run x86 ELF binaries on Alpha machines
  - flat: uClinux FLAT format binaries
  - misc: used for wrapper-driven binary formats, e.g. interpreted languages like Java, Python, .net, etc.
  - som: HP/UX binary executable format
- Execve loads the first block of the executable and calls the available binary format handlers until one is found which can execute the binary

ELF ABI

- Executable and Linkage Format Application Binary Interface (ELF ABI)
  - Data representation: byte ordering, fundamental types, alignment
  - Function calling: register usage, stack frame layout, parameter passing,
  - OS interface: virtual address space, initial registers, environment, arguments, auxiliary vector
  - Object file format: header, sections, relocations, global offset table, procedure link table
- ELF binary format handler checks for ".ELF" in the first block

```
00000000: 7f45 4c46 0202 0100 0000 0000 0000 0000  .ELF............
00000010: 0002 0016 0000 0001 0000 0000 8000 2bfc  ..............+.
00000020: 0000 0000 0000 0040 0000 0000 0001 7ed0  .......@......~.
00000030: 0000 0000 0040 0038 0009 0040 001d 001c  .....@.8...@....
00000040: 0000 0006 0000 0005 0000 0000 0000 0040  ...............@
```

- System z supplement to the ELF ABI
  www.linuxfoundation.org/spec/ELF/zSeries/lzsabi0_s390.html

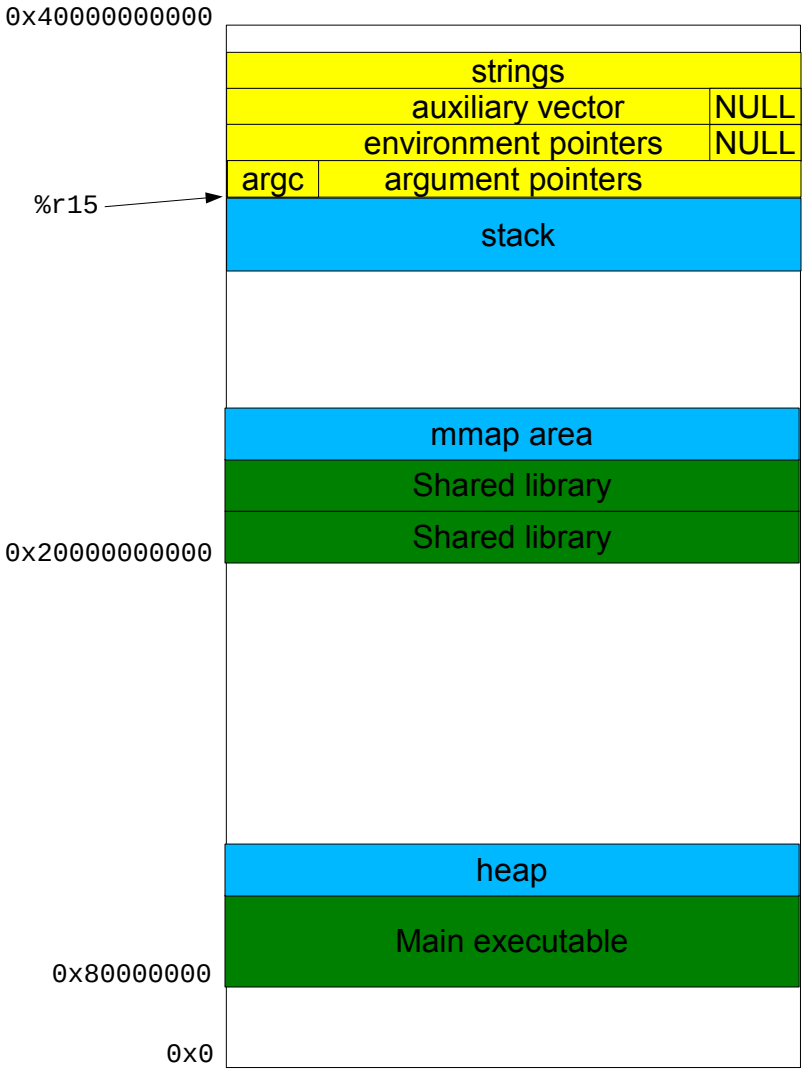
ELF ABI: registers

Register name	Usage	Call effect
%r0,%r1	General purpose	Volatile
%r2	Parameter passing and return value	Volatile
%r3, %r4, %r5	Parameter passing	Volatile
%r6	Parameter passing	Volatile
%r7 - %r11	Local variables	Saved
%r12	Local variable, GOT pointer	Saved
%r13	Local variable, literal pool pointer	Saved
%r14	Return address	Volatile
%r15	Stack pointer	Saved
%f0, %f2, %f4, %f6	Parameter passing and return values	Volatile
%f1, %r3, %f5, %f7	General purpose	Saved
%f8 - %f15	General purpose	Volatile
%a0 - %a1	Reserved for system use (TLS pointer)	Reserved
%a2 - %a15	General purpose	Volatile

ELF ABI: entries in the auxiliary vector

AT_EXECFD/AT_EXECFN	File descriptor and filename of the executable
AT_PHDR/AT_PHNUM	Program headers address and size for program
AT_PAGESZ	System page size
AT_BASE	Base address of interpreter (dynamic linking)
AT_FLAGS	Flags
AT_ENTRY	Entry point of the program
AT_UID/AT_EUID	Real and effective user id
AT_GID/AT_EGID	Real and effective group id
AT_PLATFORM	String identifying platform
AT_HWCAP	Machine dependent hints about processor capabilities
AT_FPUCW	Used FPU control word
.. and some more	

ELF ABI: System z standard virtual memory layout

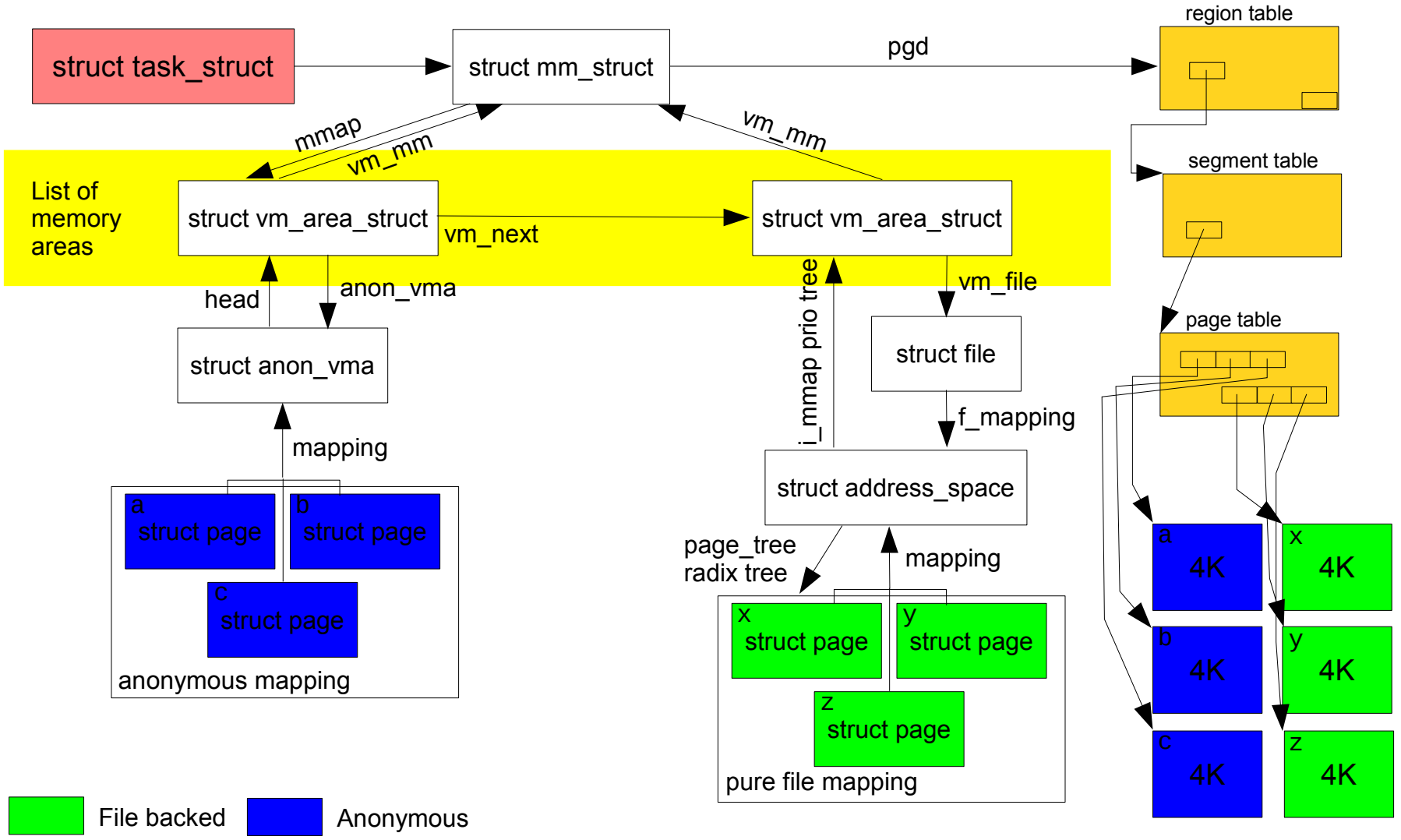


- Main executable usually at 2GB
  - Standard linker script uses 2GB as starting point
- Heap is located right after the main executable
- Libraries usually at 2TB
  - Prelink pre-allocates libraries and binaries
- Stack allocated near the end of the address space
  - Stack grows down
- Argument count: the number of argument pointers
- Argument pointers refer to a set of positional strings passed to the application. Example: `"cp" "file-a" "file-b"`
- Environment pointers refer to a set of dynamic named string values, each process has its own private set. Example: `"PWD=/home/user"`
- The auxiliary vector conveys information from the OS to the application.

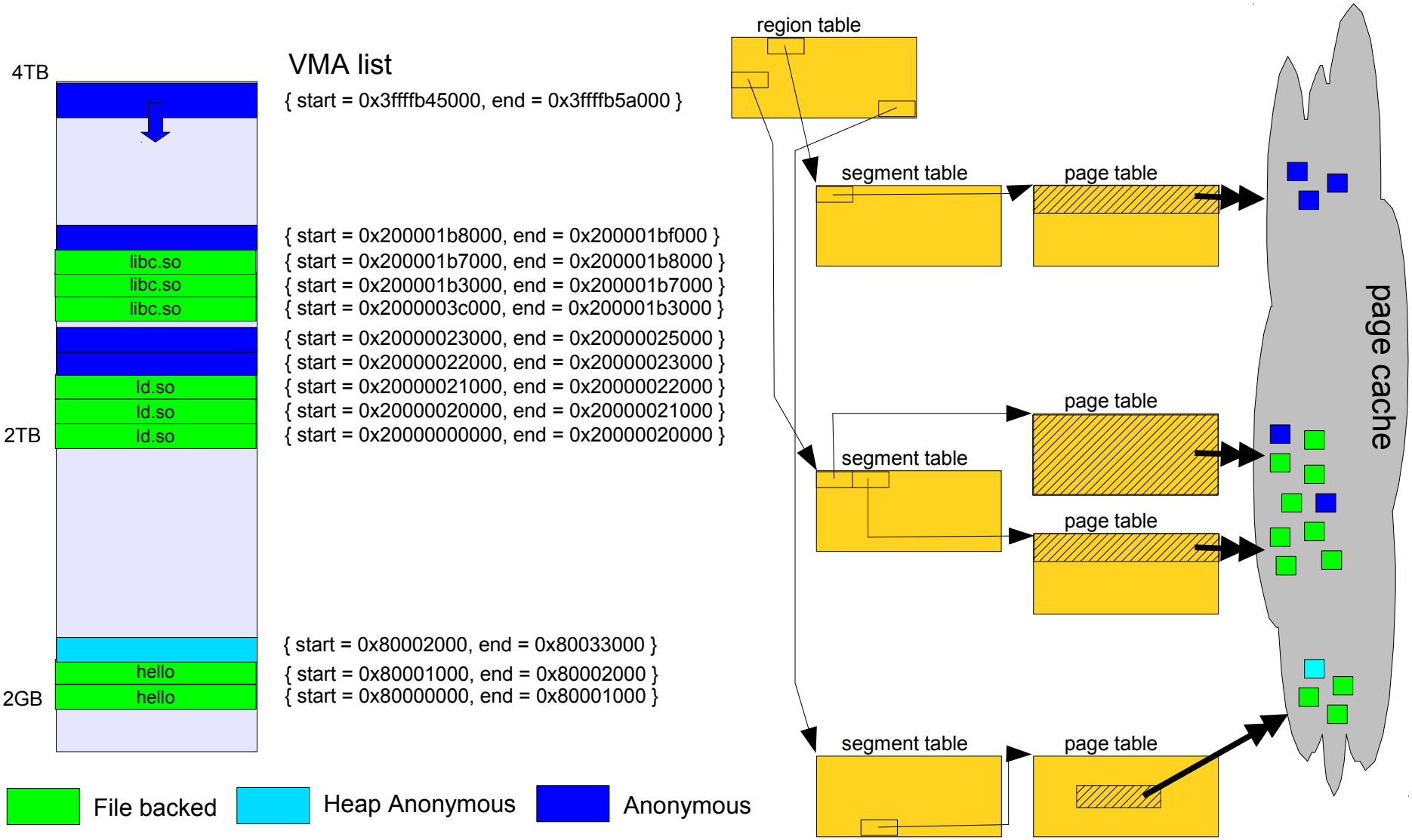
Memory management data structures

- **Virtual address space (struct mm_struct)**
  - Describes the virtual address space of a process
  - Contains the head of a list of memory areas
  - Contains counters and other information about the virtual address space
- **Memory areas (struct vm_area_struct)**
  - Defines an area in the mm_struct with start and end address
  - Either a “window” into a file starting at an offset or an anonymous area (or both)
- **Anonymous memory area (struct anon_vma)**
  - The mapping of an anonymous page points to a struct anon_vma
  - The anon_vma contains the head of a list of related memory areas
  - vmas on the list will be related by forking and vma splitting / merging
- **Page descriptor (struct page)**
  - Each physical page in the system has a struct page associated with it
  - Used to keep track of whatever the page is used for

Virtual memory areas kernel data structures

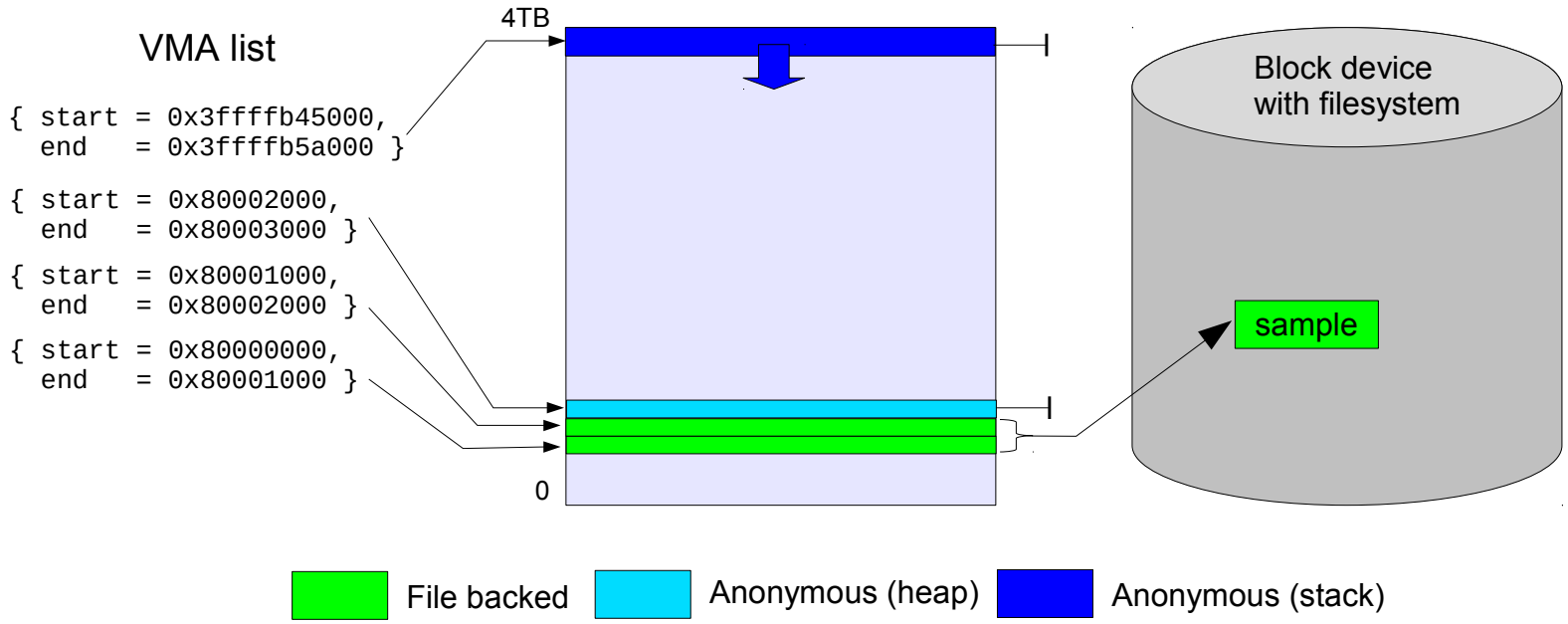


Virtual memory areas and page tables



/proc/<pid>/maps of a simple static ELF executable

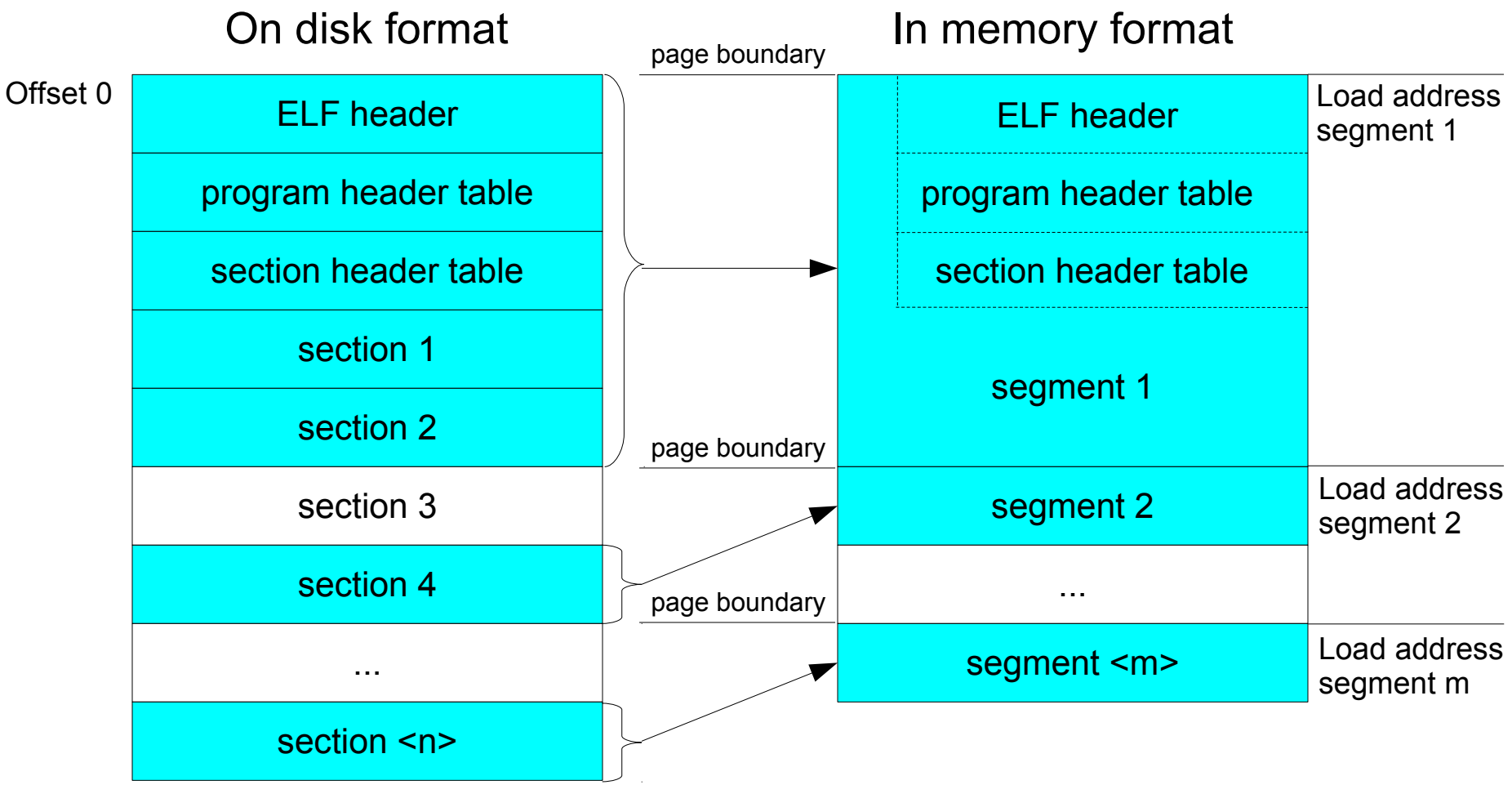
```
Start            End              Acc  Offset   Bdev  Ino     File
=============================================================================
   80000000-   80001000 r-xp 00000000 5e:12 32275   /root/sample
   80001000-   80002000 rwxp 00000000 5e:12 32275   /root/sample
   80002000-   80003000 rw-p 00000000 00:00 0       [heap]
3ffffb5f000-3ffffb74000 rw-p 00000000 00:00 0       [stack]
```



ELF object file format – header (readelf -h)

```
ELF Header:
  Magic:   7f 45 4c 46 02 02 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, big endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           IBM S/390
  Version:                           0x1
  Entry point address:               0x800000e8
  Start of program headers:          64 (bytes into file)
  Start of section headers:          448 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         3
  Size of section headers:           64 (bytes)
  Number of section headers:         8
  Section header string table index: 7
```

ELF object file format



ELF object file format – program headers (readelf -l)

```
Elf file type is EXEC (Executable file)
Entry point 0x800000e8
There are 3 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000080000000 0x0000000080000000
                 0x0000000000000150 0x0000000000000150  R E    1000
  LOAD           0x0000000000000150 0x0000000080001150 0x0000000080001150
                 0x0000000000000010 0x0000000000000020  RW     1000
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     8

 Section to Segment mapping:
  Segment Sections...
   00     .text .rodata .eh_frame
   01     .data .bss
   02
```

ELF object file format – section headers (readelf -S)

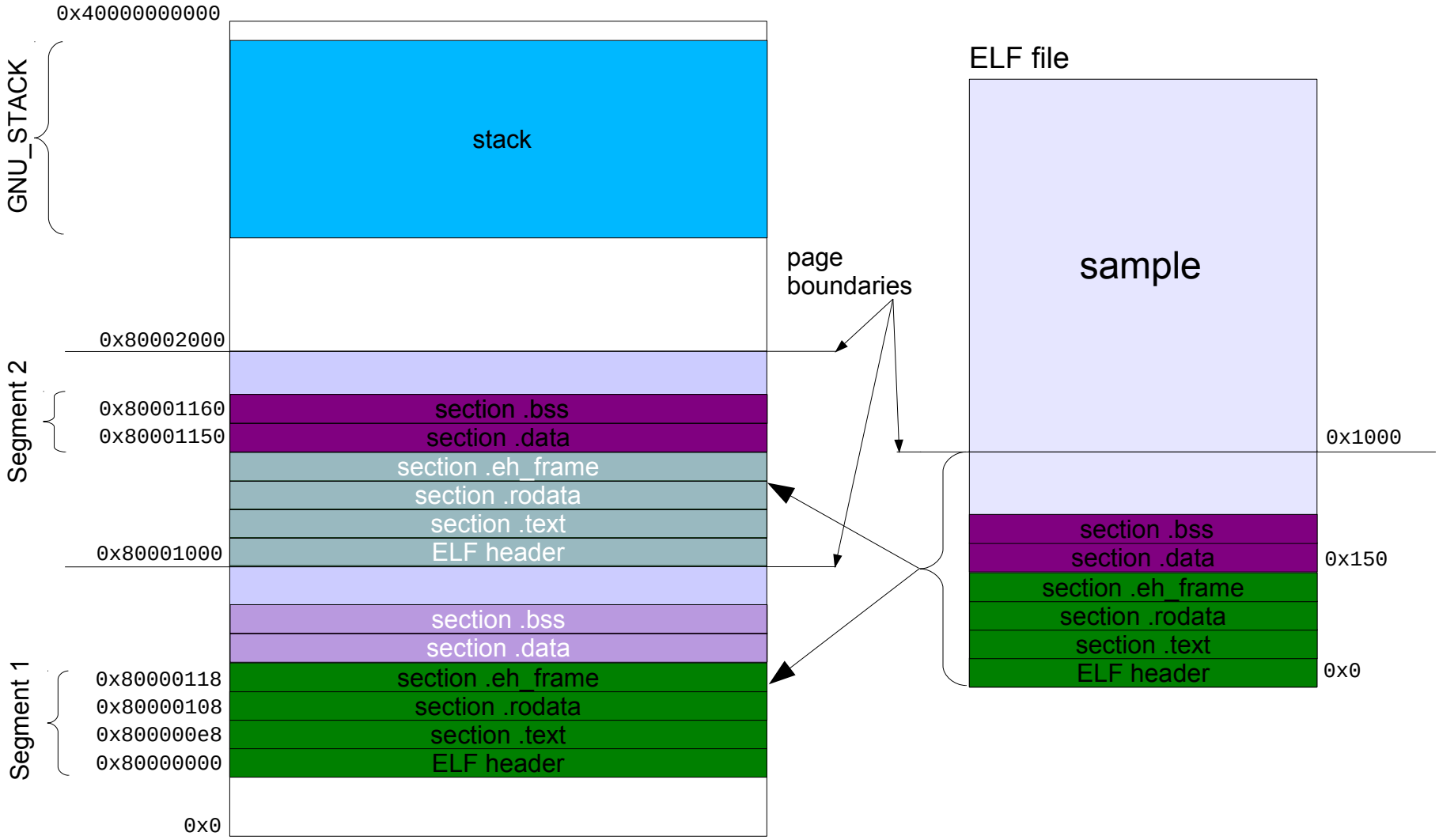
```
There are 8 section headers, starting at offset 0x1c0:

Section Headers:
  [Nr] Name              Type            Address          Offset
       Size              EntSize         Flags  Link  Info  Align
  [ 0]                   NULL            0000000000000000 00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS        00000000800000e8 000000e8
       0000000000000020  0000000000000000  AX       0     0     4
  [ 2] .rodata           PROGBITS        0000000080000108 00000108
       0000000000000010  0000000000000000   A       0     0     4
  [ 3] .eh_frame         PROGBITS        0000000080000118 00000118
       0000000000000038  0000000000000000   A       0     0     8
  [ 4] .data             PROGBITS        0000000080001150 00000150
       0000000000000010  0000000000000000  WA       0     0     4
  [ 5] .bss              NOBITS          0000000080001160 00000160
       0000000000000010  0000000000000000  WA       0     0     4
  [ 6] .comment          PROGBITS        0000000000000000 00000160
       0000000000000028  0000000000000000           0     0     1
  [ 7] .shstrtab         STRTAB          0000000000000000 00000188
       0000000000000037  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

Segment 1

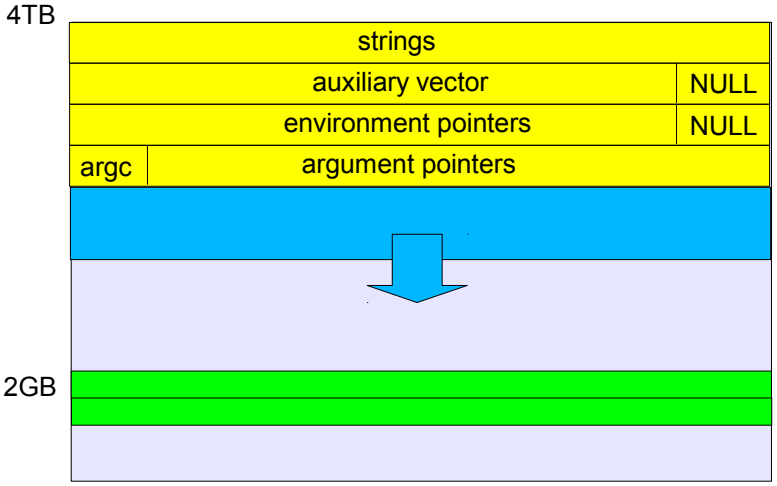
Segment 2

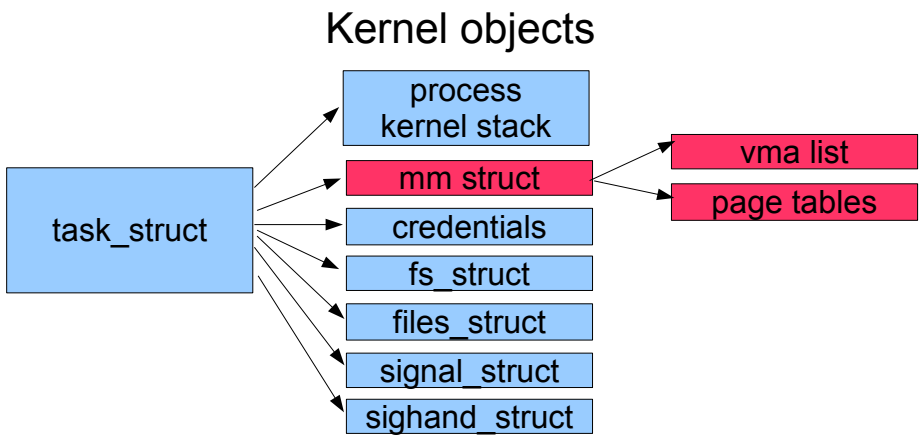
Mapping of file pages to the user address space



What happens on execve() ?

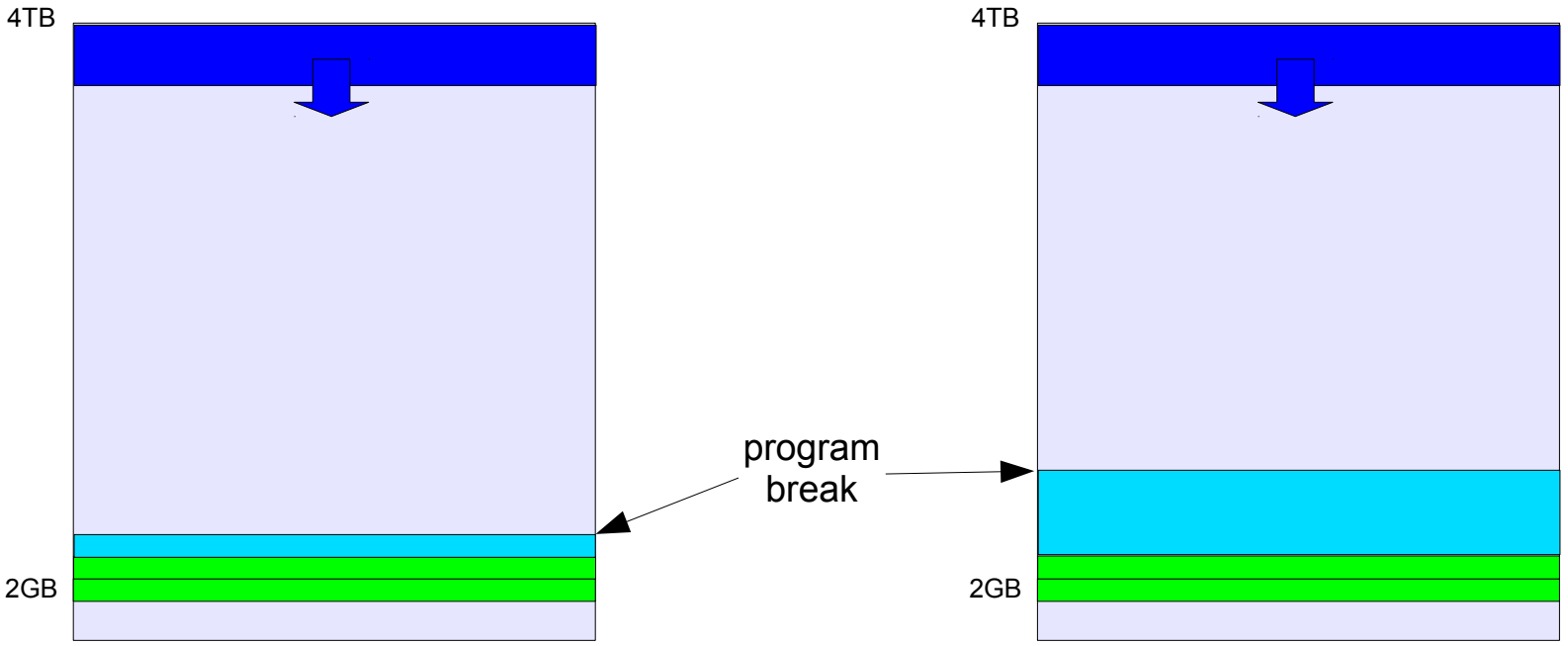
- Kernel copies environment & arguments from the current process
  - Number of internal pages used to be limited: argument list too long errors
- Kernel clears existing mappings and maps the PT_LOAD segments of the executable
  - Only the vma is created and the top level page directory is allocated
- Kernel creates auxiliary vector, environment & arguments
  - User copy causes page faults → stack page tables are created
  - Clearing of .bss section causes faults for writable segment of executable
- Kernel initializes processor state, including stack pointer & initial PSW





What happens on brk() ?

- brk/sbrk system calls change the location of the program break
- brk sets the end address of the heap, sbrk increases the size of the heap
- The new memory space is not backed with real memory right away, pages are allocated on first access



What happens on exit() / exit_group() ?

- **Almost all kernel structures for the process are freed**
  - For shared structures the reference counter is decremented
  - Examples: mm_struct, fs_struct, files_struct, signal_struct, sighand_struct ...
- **The address space (mm_struct) is cleared before it is freed**
  - munmap from address 0 to address -1
- **The task_struct is not (yet) freed**
  - The task state is set to TASK_DEAD
  - The parent is notified that the child has exited
  - The parent needs to call wait4()/waitid() to collect the exit status
  - A dead child process with wait4()/waitid() pending is a “Zombie”

Sequence of system calls for bash executing “hello”

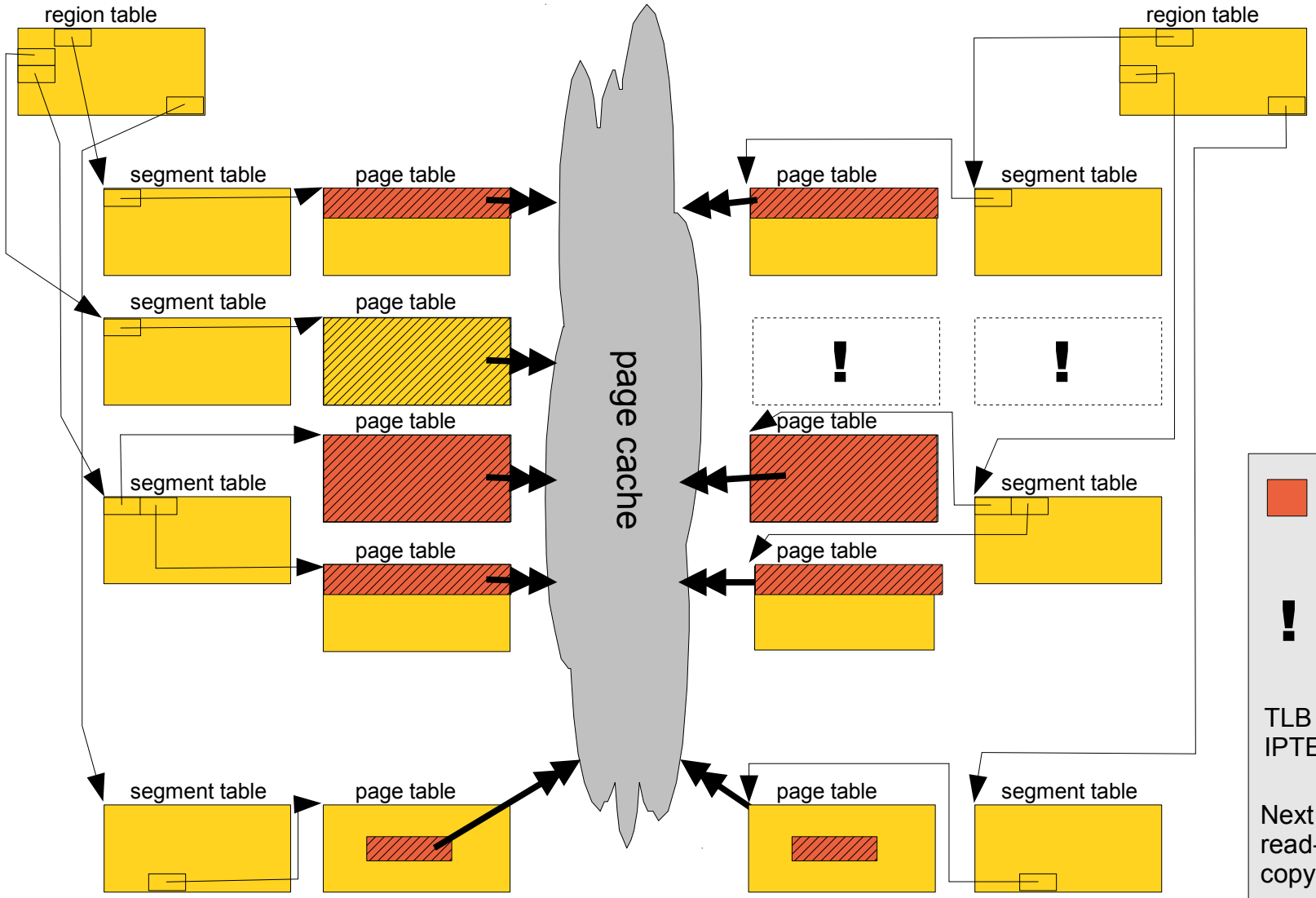
Parent	Child
`bash: clone()`	`<child process created by clone>`
`bash: wait4() <unfinished ...>`	`bash: execve(“./hello”, ...)`
	`hello: brk(0) = 0x8009d000`
	`hello: brk(0x8009df60) = 0x8009df60)`
	`hello: brk(0x800bef60) = 0x800bef60)`
	`hello: brk(0x800bf000) = 0x800bf000)`
	`hello: fstat(1, …) = 0`
	`hello: mmap(NULL, …) = 0x20000000000`
	`hello: write(1, “Hello World\n”, 12) = 12`
	`hello: exit_group(0)`
`bash: <... wait4 resumed>`	

- No “open” for file descriptor 1
  - Child inherits all open file descriptors of the parent process
  - execve closes all of them but the first 3: stdin, stdout, stderr
- No “close”, no “munmap”; exit_group takes care of that

What happens on clone() / fork() ?

- Execve() replaces the current context, fork() duplicates a context
- Allocate a new task_struct (dup_task_struct)
  - Copies almost all fields from the parent task_struct
- Create copies of various task related kernel data structures
  - Credentials, same rights for the child
  - File descriptor table, same files for the child
  - Filesystem info, same working directory, same umask
  - Copies the current set of signal handlers, empty set of pending signals
  - Duplicate virtual memory address space, only mlocks()s are not inherited
- Initialize per task information
  - Scheduler initialize per task fields and selects a run queue
  - Child gets a new pid / tpid
  - Resource utilization is reset, timers are reset
- Add new task to process tree

What happens on fork() / copy_mm() ?

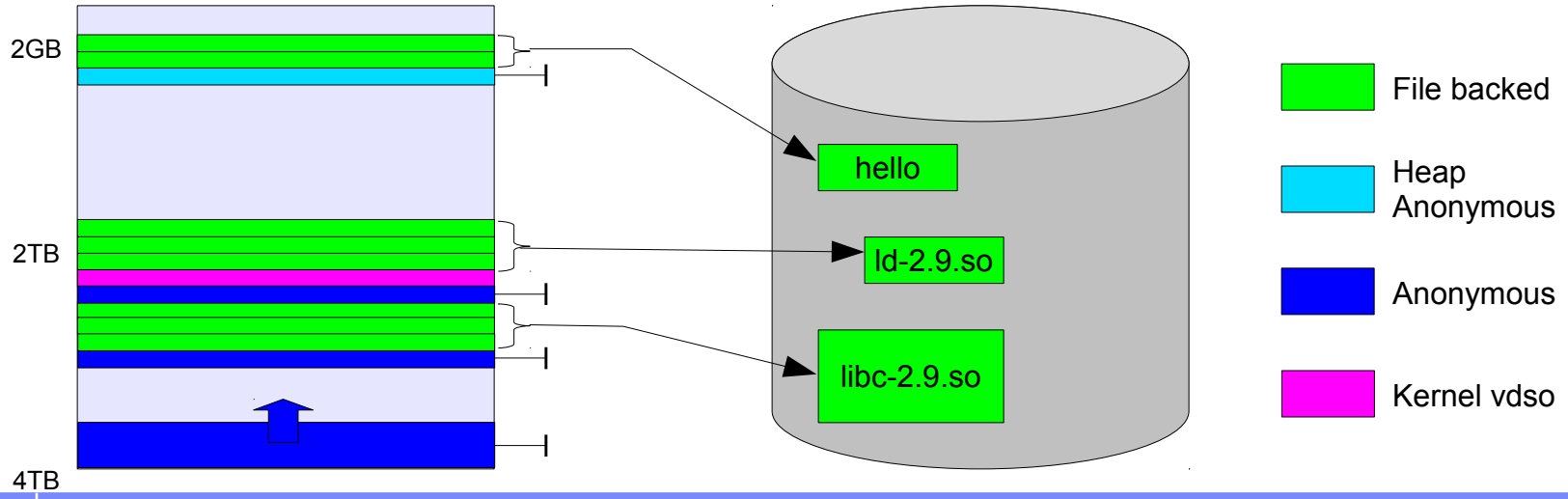


Dynamic linking of ELF objects

- **User space program can be statically or dynamically linked**
  - Dynamic ELF exec is mapped just like a statically linked exec
  - ELF .interp section gives the name of the “interpreter” = ld.so
  - ld.so is mapped and started instead of the main executable
  - ld.so loads and links the missing pieces for the dynamic executable
  - ld.so resolves the relocations between the different ELF objects
- **Usually a user space program has multiple ELF objects**
  - Main executable at 2GB (default)
  - Dynamic linker ld.so at 2TB
  - C-runtime libc.so at 2TB + sizeof ld.so object
  - More shared libraries
  - Kernel virtual dynamic shared object (vdso)
- **Shared libraries can by dynamically loaded / unloaded**
  - dlopen() / dlsym() / dlclose() calls to ld.so

/proc/<pid>/maps for dynamic “Hello World”

```
Start            End              Acc  Offset   Bdev  Ino     File
=============================================================================
   80000000-   80001000 r-xp 00000000 5e:01 11875   /root/hello
   80001000-   80002000 rwxp 00000000 5e:01 11875   /root/hello
   80002000-   80023000 rw-p 00000000 00:00 0       [heap]
20000000000-20000020000 r-xp 00000000 5e:01 272257  /lib/ld64.so.1
20000020000-20000021000 r-xp 0001f000 5e:01 272257  /lib/ld64.so.1
20000021000-20000022000 rwxp 00020000 5e:01 272257  /lib/ld64.so.1
20000022000-20000023000 rw-p 00000000 00:00 0
20000023000-20000025000 r-xp 00000000 00:00 0       [vdso]
20000025000-20000029000 rwxp 00000000 00:00 0
2000003f000-200001b6000 r-xp 00000000 5e:01 268320  /lib64/libc-2.11.90.so
200001b6000-200001ba000 r-xp 0018e000 5e:01 268320  /lib64/libc-2.11.90.so
200001ba000-200001bb000 rwxp 00192000 5e:01 268320  /lib64/libc-2.11.90.so
200001bb000-200001c1000 rwxp 00000000 00:00 0
3ffffb45000-3ffffb5a000 rw-p 00000000 00:00 0       [stack]
```



Sytem call trace of “Hello World”

bash (strace)

```
# strace -v ./hello
execve("./hello", ["./hello"], ["HOSTNAME=t6360015", "TERM=xterm", "SHELL=/bin/bash",
      "HISTSIZE=1000", "SSH_CLIENT=9.152.212.37 56750 22"..., "SSH_TTY=/dev/pts/0",
      "USER=root", "USERNAME=root", "PATH=/sbin:/bin:/usr/sbin:/bin"..., "PWD=/root",
      "SHLVL=1", "HOME=/root", "BASH_ENV=/root/.bashrc", "LOGNAME=root",
      "_=/usr/bin/strace"]) = 0
```

ld.so.1

```
brk(0) = 0x80002000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =0x20000002000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =0x20000003000
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=93073, ...}) = 0
mmap(NULL, 93073, PROT_READ, MAP_PRIVATE, 3, 0) = 0x20000005000
close(3) = 0
open("/lib64/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\2\2\1\0\0\0\0\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=10086079, ...}) = 0
mmap(NULL, 1574256, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
     0x2000001c000
mmap(0x20000193000, 20480, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE,
     3, 0x176000) = 0x20000193000
mmap(0x20000198000, 17776, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS,
     -1, 0) = 0x20000198000
close(3)                                = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
     0x2000019d000
mprotect(0x20000193000, 16384, PROT_READ) = 0
mprotect(0x20000020000, 4096, PROT_READ) = 0
munmap(0x20000028000, 93073) = 0
```

hello

```
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
     0x20000028000
write(1, "Hello world 0x80002410\n"..., 12) = 12
exit_group(0)                           = ?
```

ELF object file format – relocations (readelf -r)

```
Relocation section '.rela.dyn' at offset 0x380 contains 1 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
0000800019d8  00010000000a R_390_GLOB_DAT    0000000000000000 __gmon_start__ + 0

Relocation section '.rela.plt' at offset 0x398 contains 3 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
0000800019c0  00020000000b R_390_JMP_SLOT    000000008000044c malloc + 0
0000800019c8  00030000000b R_390_JMP_SLOT    000000008000046c printf + 0
0000800019d0  00040000000b R_390_JMP_SLOT    000000008000048c __libc_start_main + 0
```

ELF object file format – typical relocations

R_390_64/R_390_32	Direct address of the symbol 64 bit / 32 bit
R_390_GLOB_DAT	Create global offset table (GOT) entry.
R_390_JMP_SLOT	Create procedure link table (PLT) entry.
R_390_RELATIVE	Adjust by program base.
R_390_TLS_TPOFF	Negated offset in static thread local storage (TLS) block.
R_390_TLS_DTPMOD	ID of module containing symbol.

Links for further reading

- ELF and ABI standards
  http://refspecs.freestandards.org/elf/
- System z supplement to the ELF ABI
  http://www.linuxfoundation.org/spec/ELF/zSeries/lzsabi0_s390.html
- Native POSIX thread library support
  http://people.redhat.com/~drepper/nptl-design.pdf
- Thread local storage support
  http://people.redhat.com/drepper/tls.pdf
- Understanding the Linux virtual memory manager
  http://ptgmedia.pearsoncmg.com/
  images/0131453483/downloads/gorman_book.pdf

Questions?

Martin Schwidefsky

Linux on System z Development

Schönaicher Strasse 220
71032 Böblingen, Germany

Phone +49 (0)7031-16-2247
schwidefsky@de.ibm.com