

Wie der AFS Klient funktioniert

Hartmut Reuter
reuter@rzg.mpg.de

Zusammenfassung:

In dieser Darstellung wird versucht, die Funktionsweise des AFS-Klienten verständlich zu machen. Außerdem werden Debugging-Techniken für den AFS-Klienten vorgestellt.

Themen

- Virtuelle Filesysteme in Unix/Linux
- Implementation des AFS-Klienten
 - Aufbau des Source-Baumes
 - Start des AFS-Klienten
 - Einzelschritte beim Lesen eines Files
 - Einzelschritte beim Schreiben eines Files
- Debugging Hilfen
 - fstrace
 - kdump
- Tuning
 - Cache Parameter
 - Disk-Cache oder Memory-Cache oder ramdisk?
 - Direkter Zugriff auf Fileserver-Partitionen

Virtuelle Filesysteme VFS

- Einheitliche Schnittstelle zu Filesystemen
 - Wie Klasse in C++ mit Methoden, nachempfunden in C
z.B. in AIX in /usr/include/sys/vfs.h:

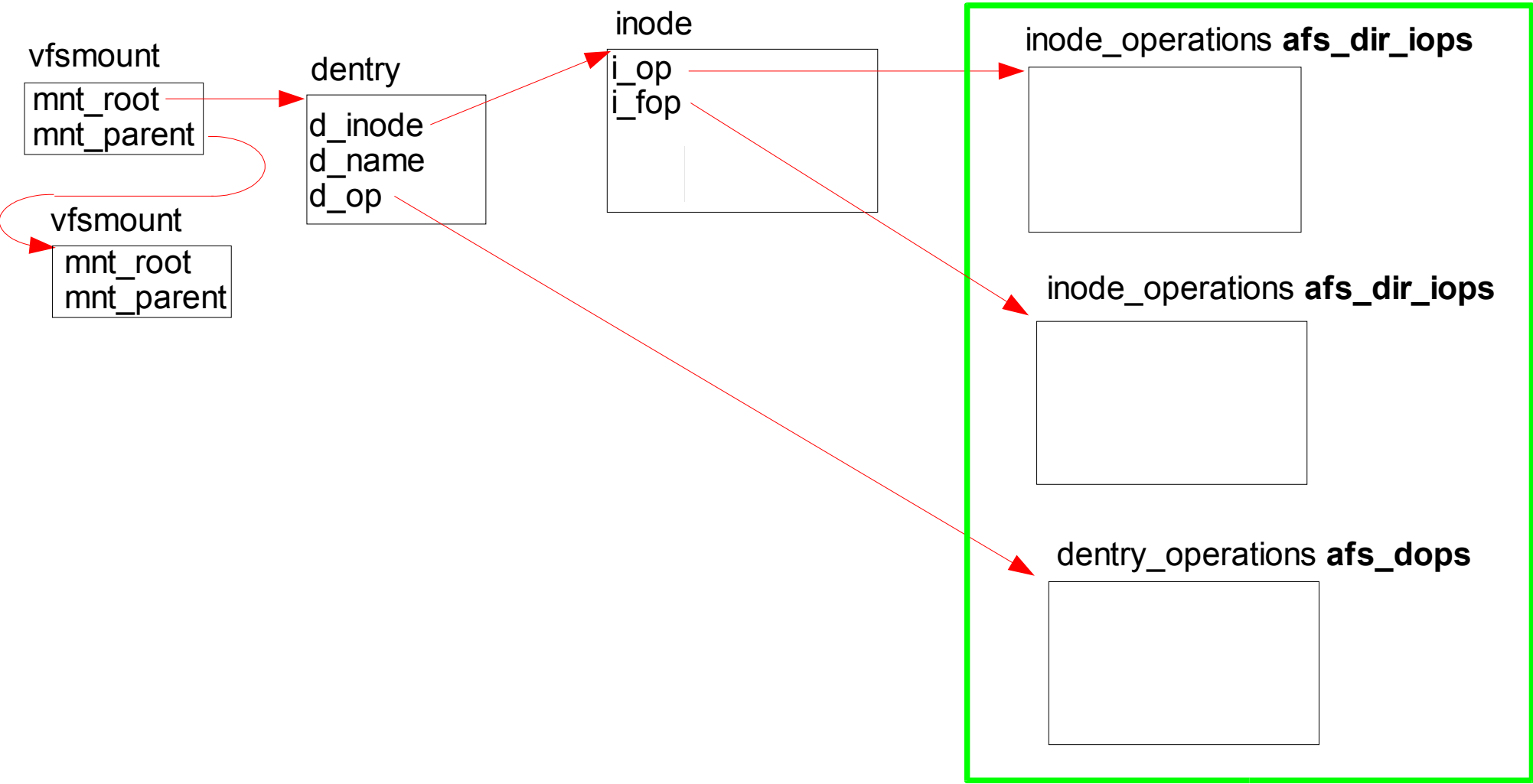
```
struct vfsops {
    int (*vfs_mount)(struct vfs *, struct ucred *);
    int (*vfs_unmount)(struct vfs *, int, struct ucred *);
    int (*vfs_root)(struct vfs *, struct vnode **, struct ucred *);
    int (*vfs_statfs)(struct vfs *, struct statfs *, struct ucred *);
    int (*vfs_sync)(struct gfs *);
    int (*vfs_vget)(struct vfs *, struct vnode **, struct fileid *,
                   struct ucred *);
    int (*vfs_cntl)(struct vfs *, int, caddr_t, size_t, struct ucred *);
    int (*vfs_quotactl)(struct vfs *, int, uid_t, caddr_t, struct ucred *);
    int (*vfs_syncvfs)(struct gfs *, struct vfs *, int, struct ucred *);
};
```

Vnodes

- Einheitliche Schnittstelle zu Objekten im Filesystem
z.B. in AIX in `/usr/include/sys/vnode.h`:

```
struct vnodeops {
    int (*vn_link)(struct vnode *, struct vnode *, char *, struct ucred *);
    int (*vn_mkdir)(struct vnode *, char *, int32long64_t, struct ucred *);
    int (*vn_mknod)(struct vnode *, caddr_t, int32long64_t, dev_t,
        struct ucred *);
    int (*vn_remove)(struct vnode *, struct vnode *, char *, struct ucred *);
    int (*vn_rename)(struct vnode *, struct vnode *, caddr_t,
        struct vnode *, struct vnode *, caddr_t, struct ucred *);
    int (*vn_rmdir)(struct vnode *, struct vnode *, char *, struct ucred *);
    int (*vn_lookup)(struct vnode *, struct vnode **, char *,
        int32long64_t, struct vattr *, struct ucred *);
    int (*vn_fid)(struct vnode *, struct fileid *, struct ucred *);
    int (*vn_open)(struct vnode *, int32long64_t, ext_t, caddr_t *,
        struct ucred *);
    int (*vn_create)(struct vnode *, struct vnode **, int32long64_t,
        caddr_t, int32long64_t, caddr_t *, struct ucred *);
    ...
};
```

VFS, hier Linux 2.4



src/afs/LINUX/osi_vnodeops.c

Einige Kontrollblöcke im AFS-Klienten

- struct vcache “Methoden” in afs/afs_vcache.c
 - AFS-vnode Information für ein File, Directory oder Link
- struct dcache “Methoden” in afs/afs_dcache.c
 - Information über ein Cache-File
- struct fcache
 - Teil von struct dcache, der in “Cacheltems” auf Platte gespeichert wird
 - Es kann mehr fcache Objekte in Cacheltems geben als dcache Blöcke im Speicher!
- struct volume “Methoden” in afs/afs_volume.c
 - nicht zu verwechseln mit struct volume in vol/volume.h
- struct fvolume
 - Teil von struct volume, der in “Volumeltems” auf Platte gespeichert wird.
- struct cell “Methoden” in afs/afs_cell.c
 - wird auch in “Cellltems” auf der Platte gespeichert
- struct server “Methoden” in afs/afs_server.c
- struct unixuser “Methoden” in afs/afs_user.c
- struct conn “Methoden” in afs/afs_conn.c

Wie sprechen Programme zum AFS Klienten

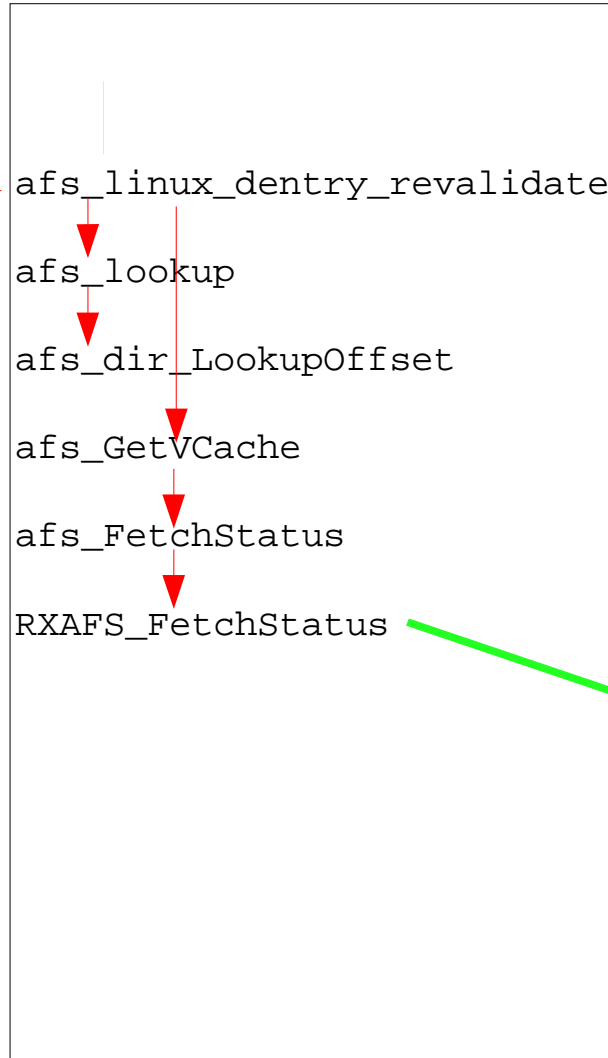
- C-library Aufrufe, die in System-Calls münden
 - open, close, read, write, readdir, stat, flock, ...
 - Alle diese Aufrufe kommen über afs/<OS>/osi_vnodeops.c in den Klienten
- piocntl
 - macht speziellen AFS-System-Call und wird in afs/afs_piocntl.c behandelt.
 - wird im Wesentlichen von "fs" benutzt
- mount, unmount, statfs
 - kommen über afs/<OS>/osi_vfsops.c rein.
- RPC-Interface
 - rxdebug, callbacks vom Server

Öffnen eines AFS-Files (in Linux)

Benutzerprog.

kernel

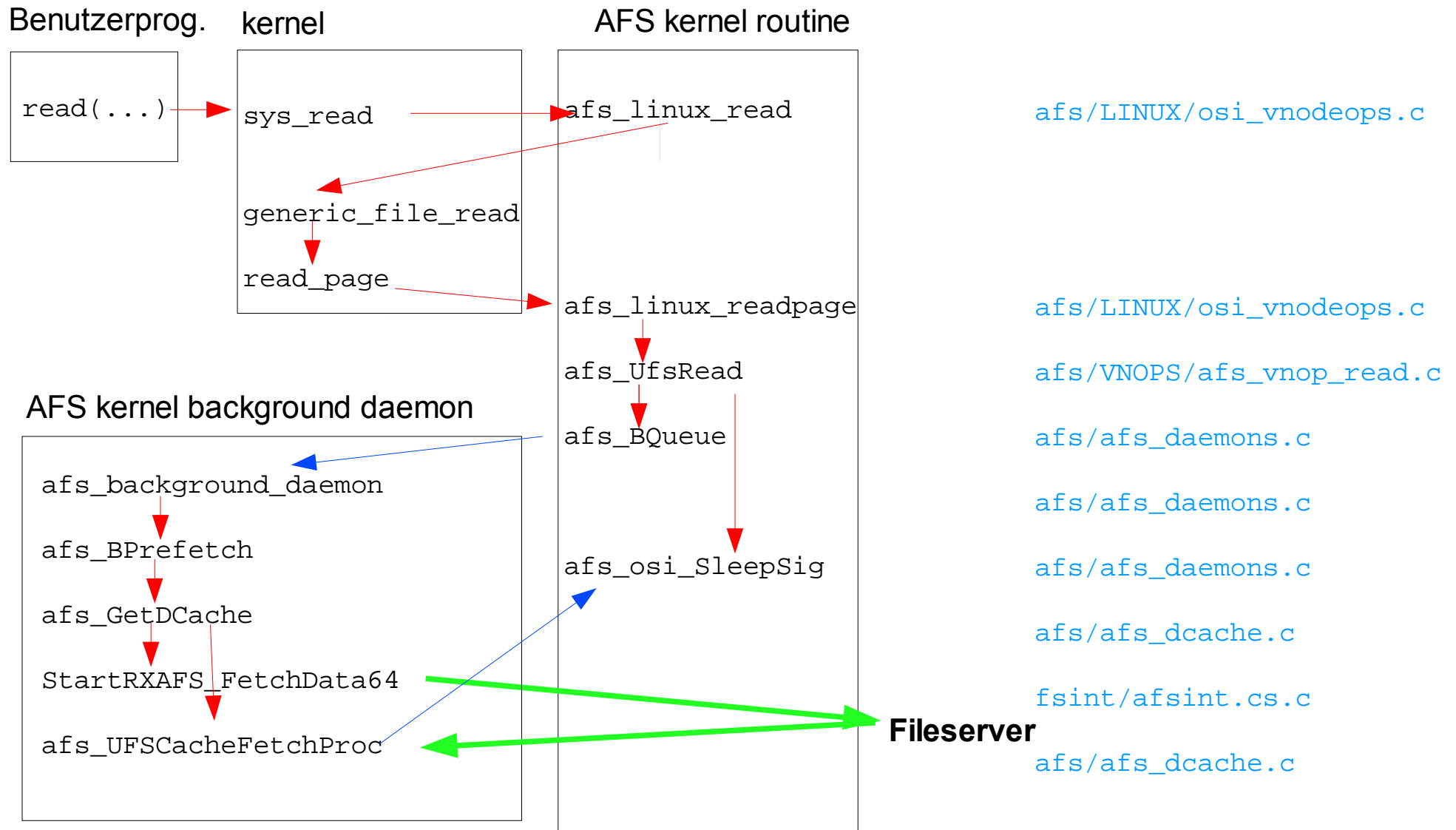
AFS kernel routines



- [afs/LINUX/osi_vnodeops.c](#)
- [afs/VNOPS/afs_vnop_lookup.c](#)
- [dir/dir.c](#)
- [afs/afs_vcache.c](#)
- [afs/afs_vcache.c](#)
- [fsint/afsint.cs.c](#)

 **Fileserver**

Lesen eines AFS-Files



Tracing im AFS-Klienten

- Der AFS-Klient kann Trace-Tables anlegen und round-robbing benutzen.
 - Definitionen usw. in `src/config/icl.h`
 - Ausführung in `src/afs/afs_call.c`
 - aufgehoben werden:
 - Timestamp
 - Prozess-id
 - Format-Nr. für das Auslesen
 - bis zu 4 Variablen (string, short, `afs_int32`, `afs_offset_t`, ptr)
 - mehrere Trace-Sets möglich
- Aktivierung und Auslesen durch "fstrace"
 -

Tracing im AFS-Klienten

- Anschalten durch Kommando “fstrace”

```
# fstrace setset cm -a  
# fstrace setlog cmfx -bufferize 8192  
# fstrace dump -follow cmfx -file /dumps/f1 -sleep 1 &
```

cm == cache manager trace

cmfx == log-name für cm

1. Aufruf aktiviert den Trace

2. Aufruf vergrößert Log-buffer von 64 KB auf 8 MB
(wichtig, sonst gehen eventuell Zeilen verloren)

3. Aufruf startet Background-Prozess zum Sammeln der Daten im File
/dumps/f1

Wichtig: “/usr/vice/etc/C/afszcm.cat” muß vorhanden sein und der Version
der Kernel-Extension entsprechen.

Tracing im AFS-Klienten 2

- Trace-Anweisungen im Source-Code (hier aus src/afs/afs_vcache.c):

```
afs_Trace3(afs_iclSetp, CM_TRACE_SIMPLEVSTAT,  
          ICL_TYPE_POINTER, avc,  
          ICL_TYPE_OFFSET, ICL_HANDLE_OFFSET(avc->m.Length),  
          ICL_TYPE_OFFSET, ICL_HANDLE_OFFSET(length));
```

“CM_TRACE_SIMPLEVSTAT” wird in src/afs/afs_trace.et definiert:

```
ec      CM_TRACE_SIMPLEVSTAT, "SimpleVStat vp 0x%lx old len (0x%x,  
0x%x) new len (0x%x, 0x%x)"
```

Der Format-String kommt nach /usr/vice/etc/C/afszcm.cat

Tracing im AFS-Klienten 3

Beispiel für Trace-Ausgabe:

```
time 182.151737, pid 3612: Access vp 0xe50ec600 mode 0x40 len (0x0, 0x1000)
time 182.151737, pid 3612: d_delete inode 0xe5137e00 d_name afs/afs_analyze.c
time 182.151737, pid 3612: Access vp 0xe50ec600 mode 0x40 len (0x0, 0x1000)
time 182.151737, pid 3612: Setattr vp 0xe5137e00 mask 0x41 newlen (0x0, 0x0) oldlen
(0x0, 0x60d2)
time 182.151737, pid 3612: RXAFS_StoreStatus vp 0xe5137e00 len (0x0, 0x60d2)
time 182.151737, pid 3612: osi_rxSleep() at /opt/openmrafs/openafs/src/rx/rx_rdwr.c
line 228
time 182.151737, pid 2519: osi_rxWakeup() at /opt/openmrafs/openafs/src/rx/rx.c line
3256
time 182.151737, pid 3612: Analyze RPC op 5 conn 0xcb472b40 code 0x0 user 0x41656166
time 182.151737, pid 3612: Was here: /opt/openmrafs/openafs/src/afs/afs_vcache.c line
1434, code = 0
time 182.151737, pid 3612: SimpleVStat vp 0xe5137e00 old len (0x0, 0x60d2) new len
(0x0, 0x60d2)
time 182.151737, pid 3612: d_delete inode 0xe5137e00 d_name afs/afs_analyze.c
time 182.151737, pid 3612: Access vp 0xe50ec600 mode 0x40 len (0x0, 0x1000)
time 182.151737, pid 3612: d_delete inode 0xe5138000 d_name afs/afs_axscache.c
```

kdump

kdump dumpt die Variablen der AFS-kernel-extension formatiert nach stdout. Die Ausgabe kann etwas länglich sein, deshalb in File umlenken.

Hier der vcache-Eintrag, der im fstrace vorkam aus kdump:

```
e5137e00: refC=1, pv=899, pu=56904, flushDv=0.0, mapDV=0.0, truncPos=(0xffffffff,
0xffffffff),
    callb=xc70704e0, cbE=1063622604, opens=0, XoW=0, flcnt=0, mvstat=0
    states=x400, dchint=0, anyA=0x9
    quick[dc=0, stamp=0, f=0, min=0, len=(0x0, 0x0)]
    mstat[len=(0x0, 0x60d2), DV=0.1, Date=1057103589, Owner=4004, Group=4132,
Mode=0100644, linkc=1]
    lock [wait=0 excl=0 readers=0 #waiting=0 last_reader=0 writer=0 src=20]
    fid(c=2, v=536985837, n=14332, u=56926)
    Access Link list: d28c1120
        d28c1120: 0) uid=0x41656166, access=0x7f, next=0
    Callbacks queue prev= 0 next= 0
    vlruq.prev=e5138114, vlruq.next=e5137d14
    flushcnt=0, mapcnt=0
```

Enpässe

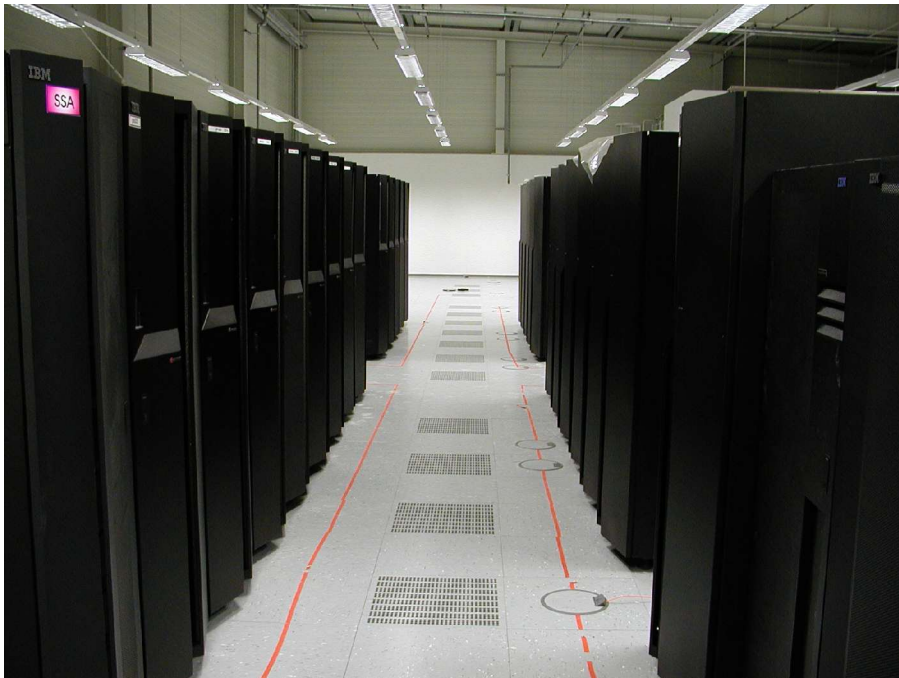
- Im Highend-Bereich ist AFS leider langsamer als NFS. Was kann man verbessern?
 - Bei Vielprozessor-Maschinen ist AFS_GLOCK() ein Problem
 - besseres Multithreading durch feineres Locking wünschenswert

fs import

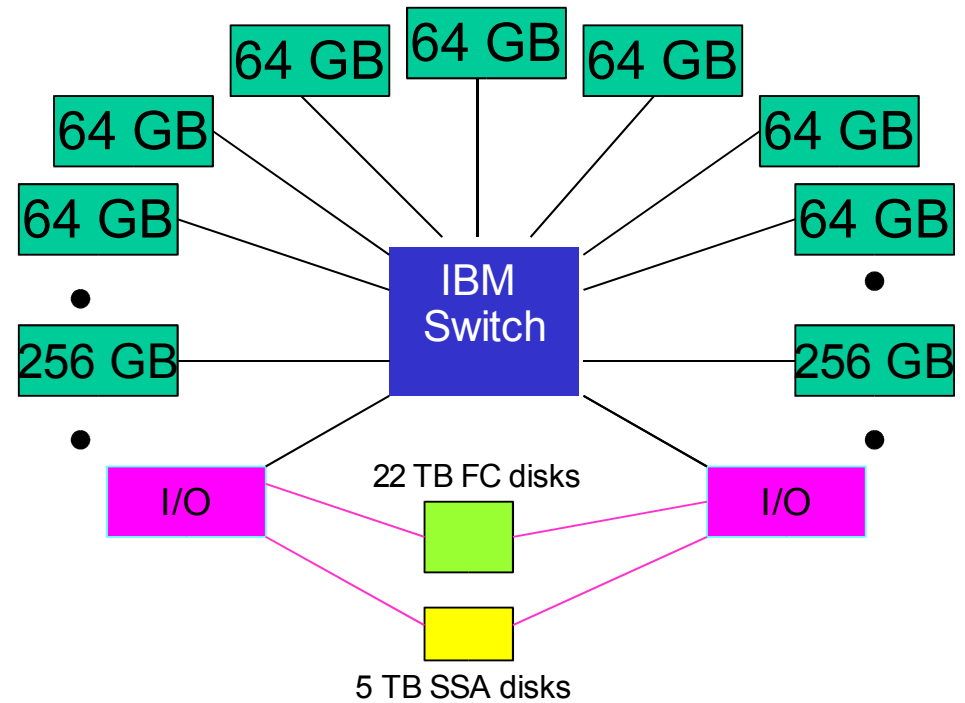
- Langlaufende Batch-Jobs sollten von AFS unabhängig sein.
 - Output in eine Subdirectory einer AFS-Fileserver-Partition, hier shared residency /vicepm (GPFS unter TSM-HSM)
 - fs import erlaubt, das File nachträglich ins AFS zu übernehmen.

```
/r/h/hwr: ls -l /r
lrwxrwxrwx    1 root      system          9 Sep  2 14:29 /r -> /vicepm/r
/r/h/hwr: df -k /vicepm
Filesystem    1024-blocks      Free %Used      Iused %Iused Mounted on
/dev/hsmgpfs  3292001280 3147224320    5%      15029      1% /vicepm
/r/h/hwr: ls -l raid5.tar
-rw-r-----  1 hwr          rzs          28417024000 Oct  5 16:22 raid5.tar
/r/h/hwr: /afs/ipp/bin/fs import raid5.tar /afs/ipp/u/hwr/test/r
file successfully imported into AFS
/r/h/hwr: cd ~/test/r
~/test/r: ls -l raid5.tar
-r-----  1 hwr          rzs          28417024000 Oct  5 16:22 raid5.tar
~/test/r: fs getres -v raid5.tar
raid5.tar: hsmgpfs
  Residency 8192: (<partition>.67109078.536993909) (536993909.214.1436)
    /vicep<x>/AFSIDat/p=/p=S+U/+//K1++2kN=
~/test/r:su
root's Password:
# ls -l /vicepm/AFSIDat/p=/p=S+U/+//K1++2kN=
-----x    1 12          system    28417024000 Oct  5 16:22 /vicepm/AFSIDat/p=/p=S
+U/+//K1++2kN=
#
```

IBM p690 4 TFlop/s, 2 TB RAM



Colony Switch based, migration to
Federation Switch starting Sep 03



24 compute nodes, 2 I/O nodes
file systems:
/u (gpfs), /ptmp (gpfs), /afs (afs)
/vicepm (gpfs with tsm-hsm)

AFS and shared filesystems

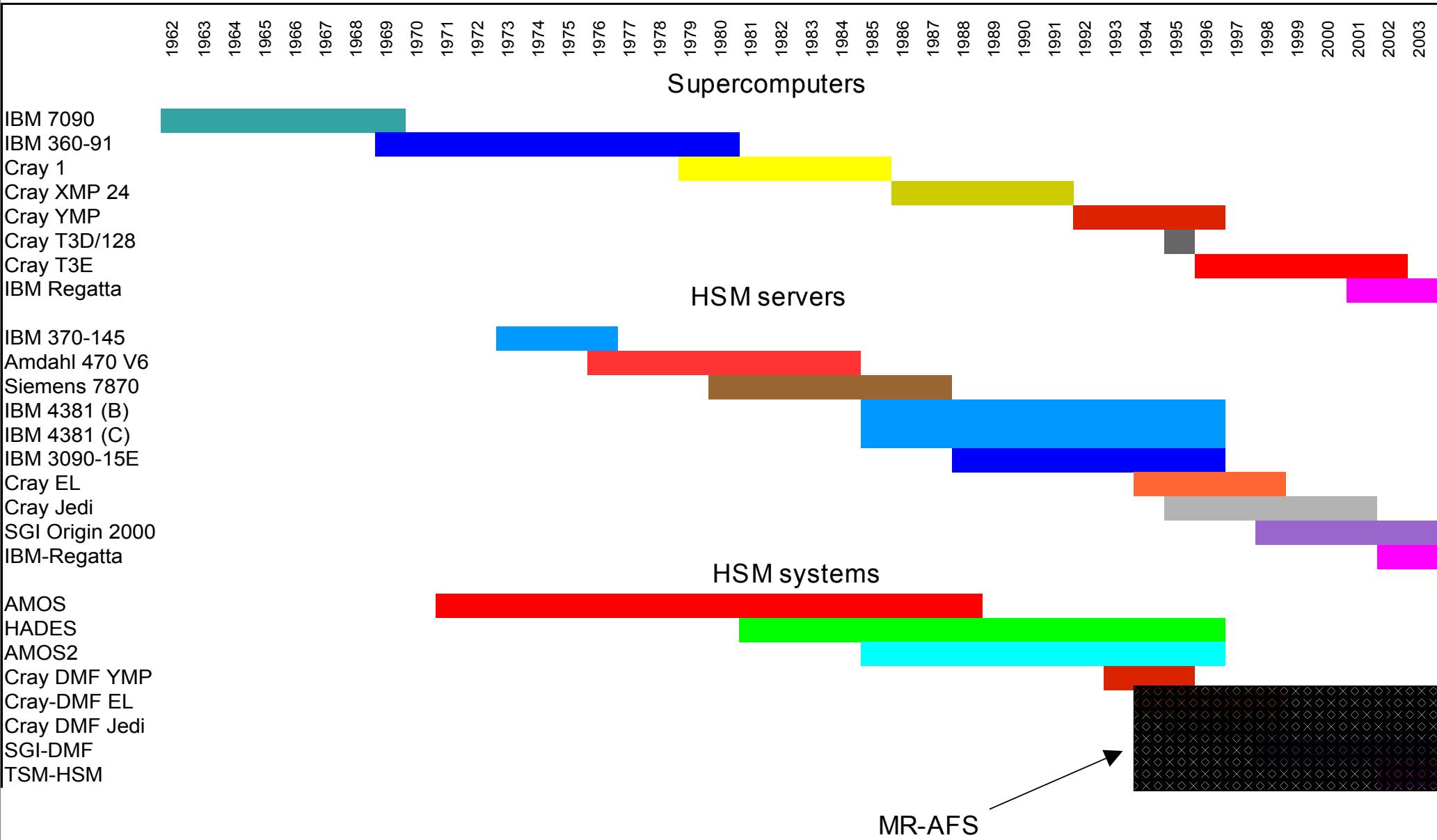
- Idea:
 - SAN-filesystems such as QFS, GPFS, CXFS and others are much faster than AFS
 - If client can „see“ fileservers partitions, he could do all I/O directly bypassing cache and network layers.
 - Put /vicep into SAN-filesystem!
- Realization:
 - Client and MR-AFS fileserver extensions to read and write /vicep directly are developed and in place.
 - Tests on Regatta show native GPFS performance in AFS.

```
~/test/r: ~hwr/afs/@sys/write_test 1GB 0 1000000000
1 writing of 104857600 bytes took 0.703 sec. (145603 Kbytes/sec)
2 writing of 104857600 bytes took 0.616 sec. (166260 Kbytes/sec)
3 writing of 104857600 bytes took 0.924 sec. (110830 Kbytes/sec)
4 writing of 104857600 bytes took 1.054 sec. (97117 Kbytes/sec)
5 writing of 104857600 bytes took 0.958 sec. (106873 Kbytes/sec)
6 writing of 104857600 bytes took 0.989 sec. (103571 Kbytes/sec)
7 writing of 104857600 bytes took 0.985 sec. (104005 Kbytes/sec)
8 writing of 104857600 bytes took 0.961 sec. (106508 Kbytes/sec)
9 writing of 104857600 bytes took 0.891 sec. (114942 Kbytes/sec)
write of 1000000000 bytes took 8.676 sec.
close took 0.000 sec.
Total data rate = 112557 Kbytes/sec. for write
~/test/r: pwd
/afs/ipp-garching.mpg.de/home/h/hwr/test/r
~/test/r: df -k /vicepm
Filesystem      1024-blocks      Free %Used      Iused %Iused Mounted on
/dev/hsmgpfs    3292001280 3274890496      1%          36      1% /vicepm
~/test/r:
```

AFS and shared filesystems

- Advantages over direct use of SAN-filesystems:
 - Files are visible on any AFS-client
 - Uids on trusted HPC-nodes may differ (coupling of remote systems)
 - Data protected by Kerberos authentication.

IPP supercomputers and HSM systems



DEISA

Distributed European Infrastructure for Supercomputing Applications

<http://www.deisa.org>

RZG participation

Major responsibilities:

- global file system(s)
AFS / GPFS / AVAKI
1 Stelle in Garching zu besetzen !!!
- DEISA applications from materials sciences and energy/fusion
research (turbulence)