„Unix is simple, but it takes a genius to understand the simplicity.“ — Dennis Ritchie

Florian Westphal && Hagen Paul Pfeifer

Vortragsuebersicht

1. Übersicht
2. IRQs
3. Speicherallokationsfunktionen (`sys_brk(2) && sys_mmap(2)`)
4. Page Faults (Seitenfehler)

Was ist der Inhalt dieses Vortrages?

1. ein genaues Verständniss von zeitlichen Zusammenhängen des Betriebssytemes

2. einen Wissensschub für Systemprogrammierer (aber auch Anwendungsentwickler). Warum sollte ich eine Datei lieber sequentiell bearbeiten als komplett wahlfrei? . . .

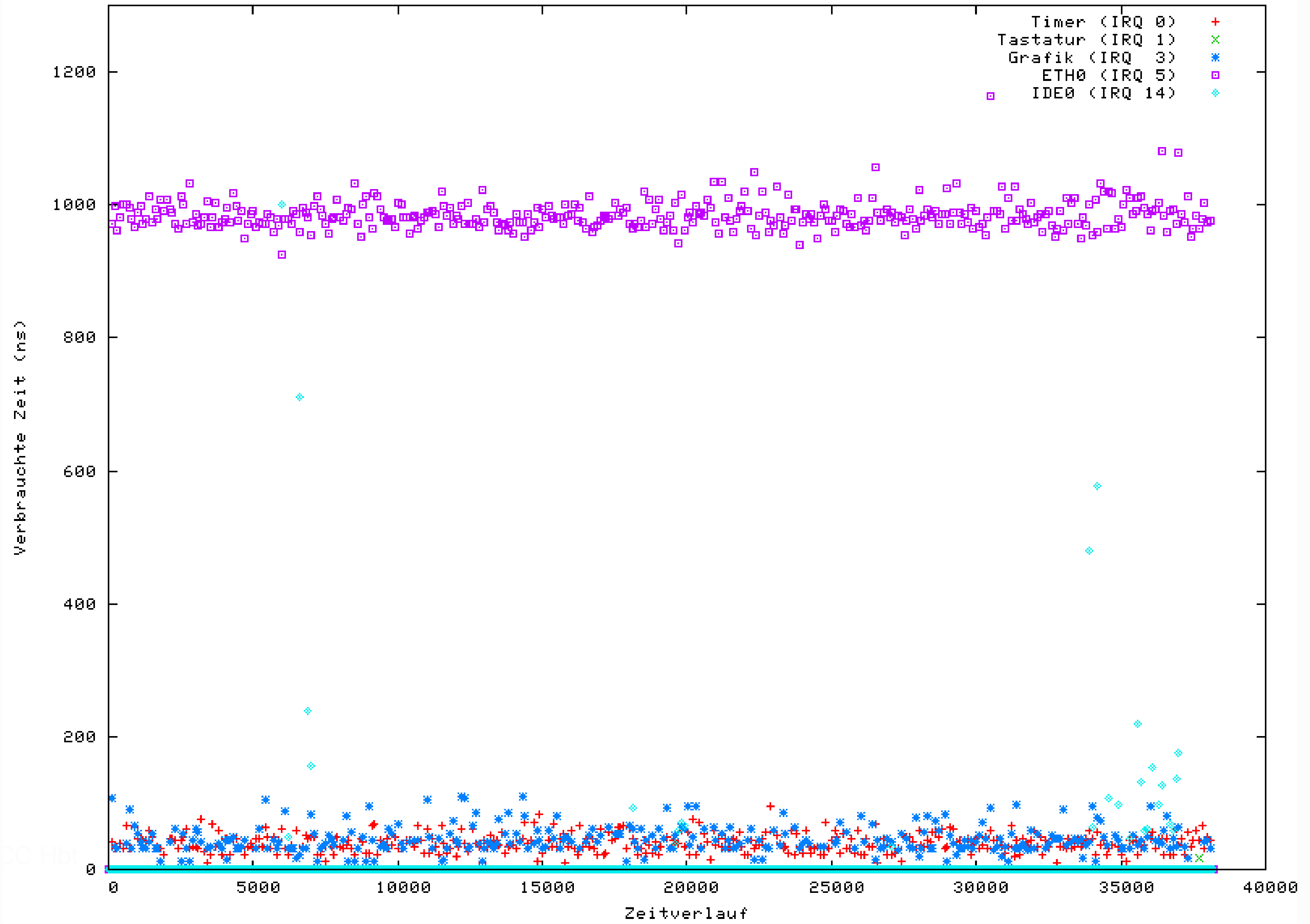
3. Viele bunte Bilder! (Oder warum haben ca. 50% der Zeitschriften Bilder zum Inhalt – oder marginalen Textanteil? Eventuell eine gewisse Ignoranz oder Angst der Leser, komplex anmutende Sachverhalte nachzuvollziehen!?)

IRQ

- Implementierung: `arch-`spezifisch
- `arch/i386/kernel/irq.c`
  - `do_IRQ()`, IRQ-Stack, ...
- `entry.S` enthält u.a.:
  - Makros zum sichern/wiederherstellen der Register
  - CPU-Exception-Handling (Division durch 0, ...)
- Akzeptable Interrupts:
  `include/asm-i386/mach-default/irq_vectors.h`

IRQ-Management

- Interface: `kernel/irq/manage.c`
- Treiber registriert Handler via `request_irq()`
  - z.B. RTC, Tastatur, Floppy, ...
- Intern:
  - allozieren von `struct irqaction`
  - `setup_irq()` registriert Handler && ggf. irq als Entropiequelle
- Freigabe mit `free_irq()`



Interrupts: ping -f
Timer (IRQ 0)
Tastatur (IRQ 1)
Grafik (IRQ 3)
ETH0 (IRQ 5)
IDE0 (IRQ 14)
Verbrauchte Zeit (ns)
Zeitverlauf
0
200
400
600
800
1000
1200
5000
10000
15000
20000
25000
30000
35000
40000

Page Faults

- Pagefault Handler wird bei Bootstrapping registriert (`arch/i386/kernel/entry.S`)
- Hardware Pagefault `arch/i386/mm/fault.c:do_page_fault()`

  1. Addresse wird beschaft (`movl \%cr2,`)
  2. Unterscheidung ob im KS (vmalloc, etc.)
  3. Nach VMA suchen (`find_vma()`)
  4. Generische Funktion: `mm/memory:handle_mm_fault()`)

- `mm/memory.c:handle_mm_fault()`

1. `pmd_alloc()` (Beachten: PAE mode)
2. `pte_alloc_map()`

- `mm/memory.c:handle_pte_fault()`

1. Demand Allocation (`do_no_page()`)
2. Demand Pagin (`do_anonymous_page()`)
3. Swap-In (`do_swap_page`)
4. Nicht lineares mapping (`do_file_page`)
5. COW (`do_wp_page()`)

Readahead

- Gemeinsamkeiten: große Textdatei & großes Programm:
  nicht jedes Segement wird immer benötigt → erst in den Speicher wenn benötigt (das war das was gerade erklärt wurde: paging! ;-)

- ABER: oft werden Abschnitte sequentielle angesprochen:

  1. `.text` Segment
  2. Logfile mit dem Lieblingseditor lesen

- ALSO: warum nicht Speicher vorlesen (cachen)(es scheint ja das mit großer Wahrscheinlichkeit er bald benötigt wird)

- WICHTIG: Suchoperationen des Festplatten Schreib- und Lesekopfes sind extrem teuer! Wenn er also schon mal zu Besuch ist soll er mehr als `PAGE_SIZE` lesen

- TIPP: um eine Datei in den Buffer Cache zu schieben → `cat big_db` (das ist für Benchmarks wichtig, dann passt auch der wahlfreie Zugriff)

mmap(2)

- whatis mmap: `map or unmap files or devices into memory`
- Bei `!MAP_ANONYMOUS` → dateispezifischen Funktionen aufrufen
- Ausflug: Was macht exec()? mmaped pages (`fs/binfmt_elf.c:elf_map`) korrigiert PAGE Attribute und setzt den %EIP auf Entry Point (naja ein `bisschen` fehlt noch, besonders wenn ein Interpreter (Link Editor) Verwendung findet ;-)
  `readelf --dynamic /usr/bin/xcruiser`
  1. `PROT_EXEC`
  2. `PROT_READ`
  3. `PROT_WRITE`

1. `MAP_PRIVATE`
2. `MAP_FIXED`
3. `MAP_DENYWRITE`
4. Beispiel aus den Mapping für den LinkEditor

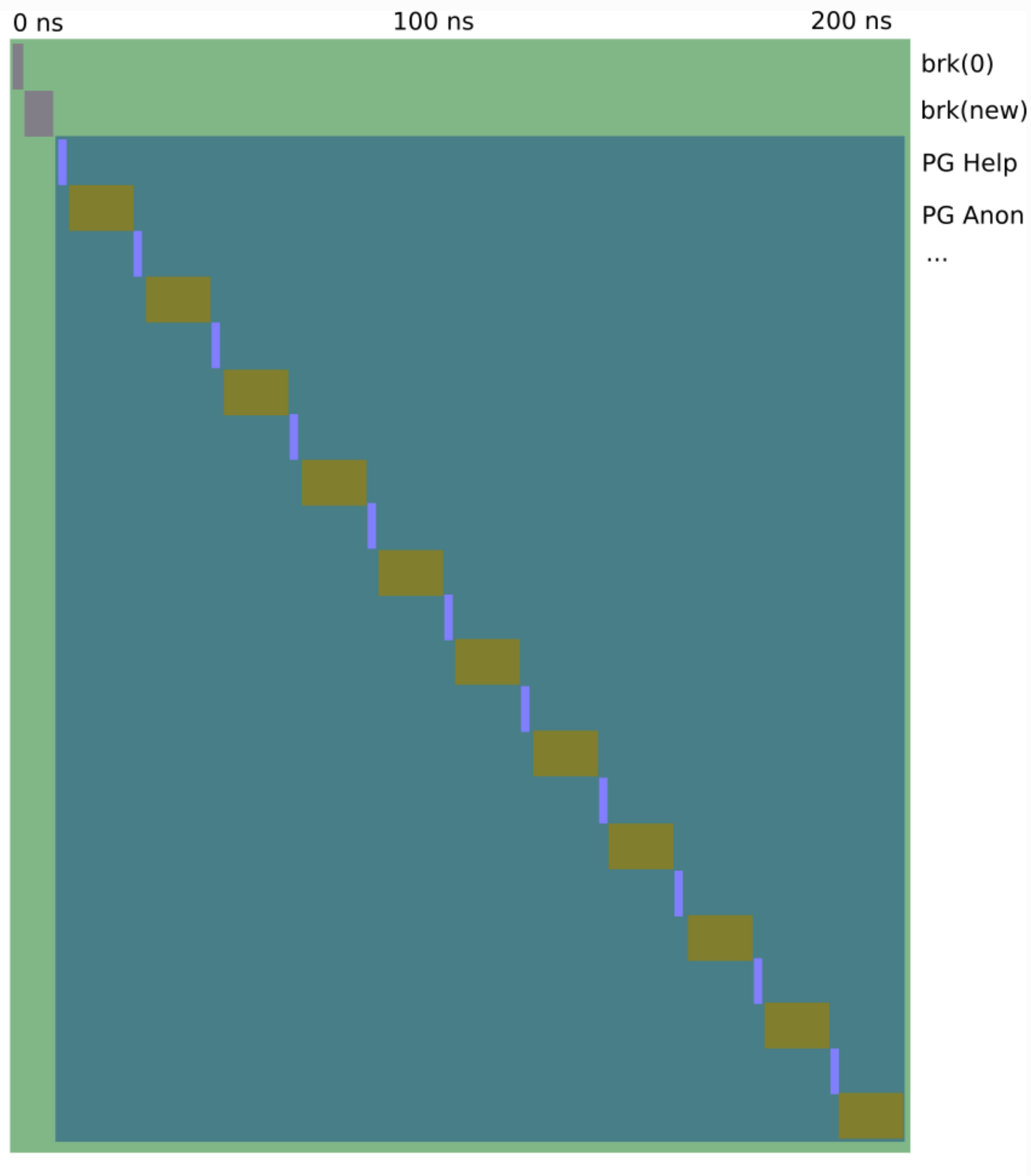
- Anmerkung: mit diesem Wissen ausgestattet ist es `fast` möglich ein triviales ABI zu entwerfen **und** zu implementieren (oder ein Userspace ELF loader (ähnlich einer Filesystem Sandbox)!

brk(2) versus mmap(2)

- glibc malloc: dlmalloc
- brk()/mmap allokieren `PAGE_SIZE` große Chunks -> Userspace Implementierung muss sich um fragmentierung kümmern
- new() ist malloc() welcher auf brk()/mmap() fußt
- `/proc/<pid>/{maps,smaps}` als zentrale Informationsstelle für Speichermappings
- Apropos Mappings: VDSO
  „userland gettimeofday“ (powerpc{32,64})
  Fallback auf syscall implementiert (z.B. interrupt latenz)

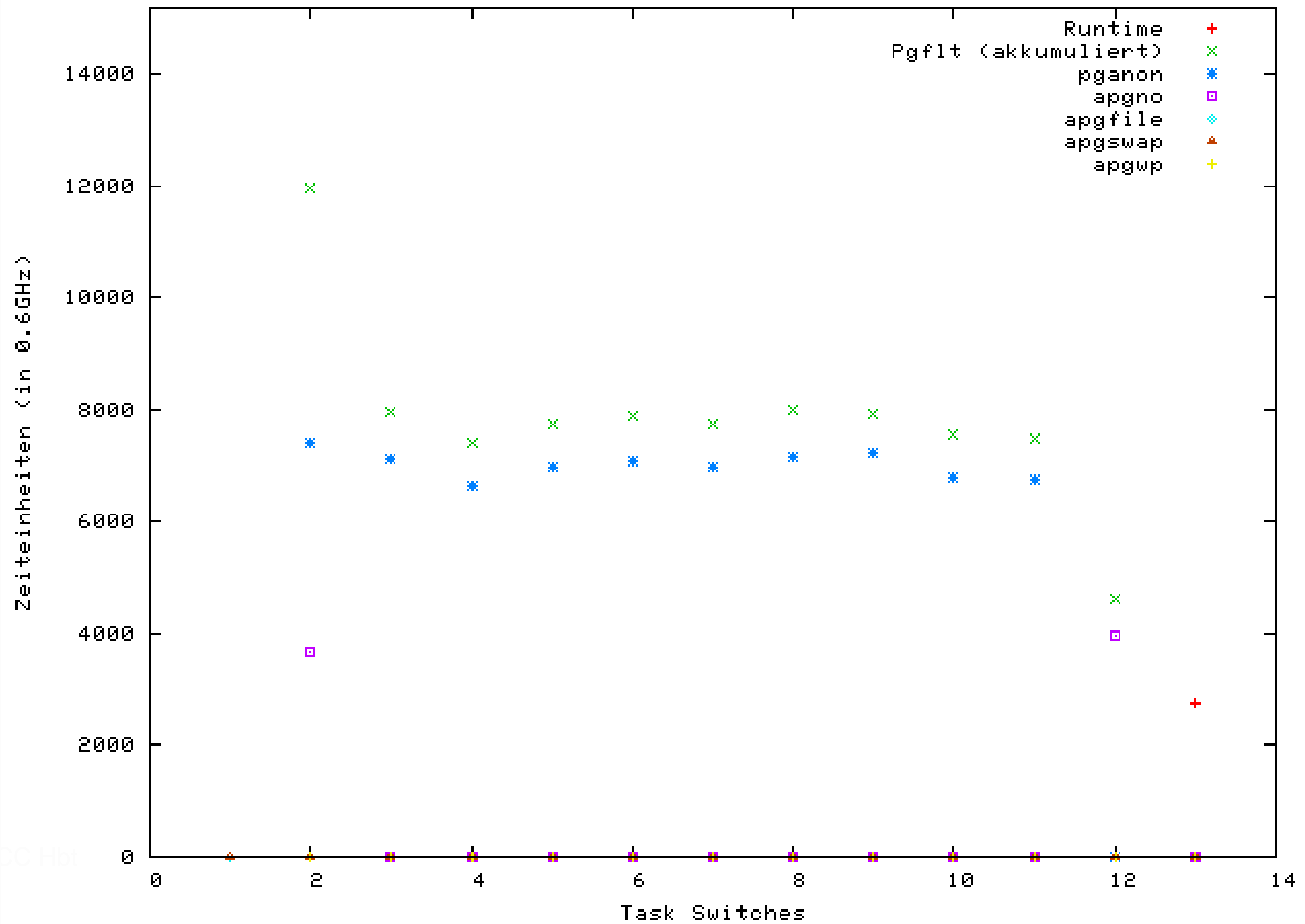
brk(2) versus mmap(2)

```
[...]
ptr = brk(0);
ptr = brk(ptr + PAGE_SIZE * ALLOC_MULT);
o_ptr = ptr - PAGE_SIZE * ALLOC_MULT;
for (k = 0; k < ALLOC_MULT; k++) {
  memset((o_ptr + ((PAGE_SIZE * (k + 1))) - 10),
          0, sizeof(int));
}
```

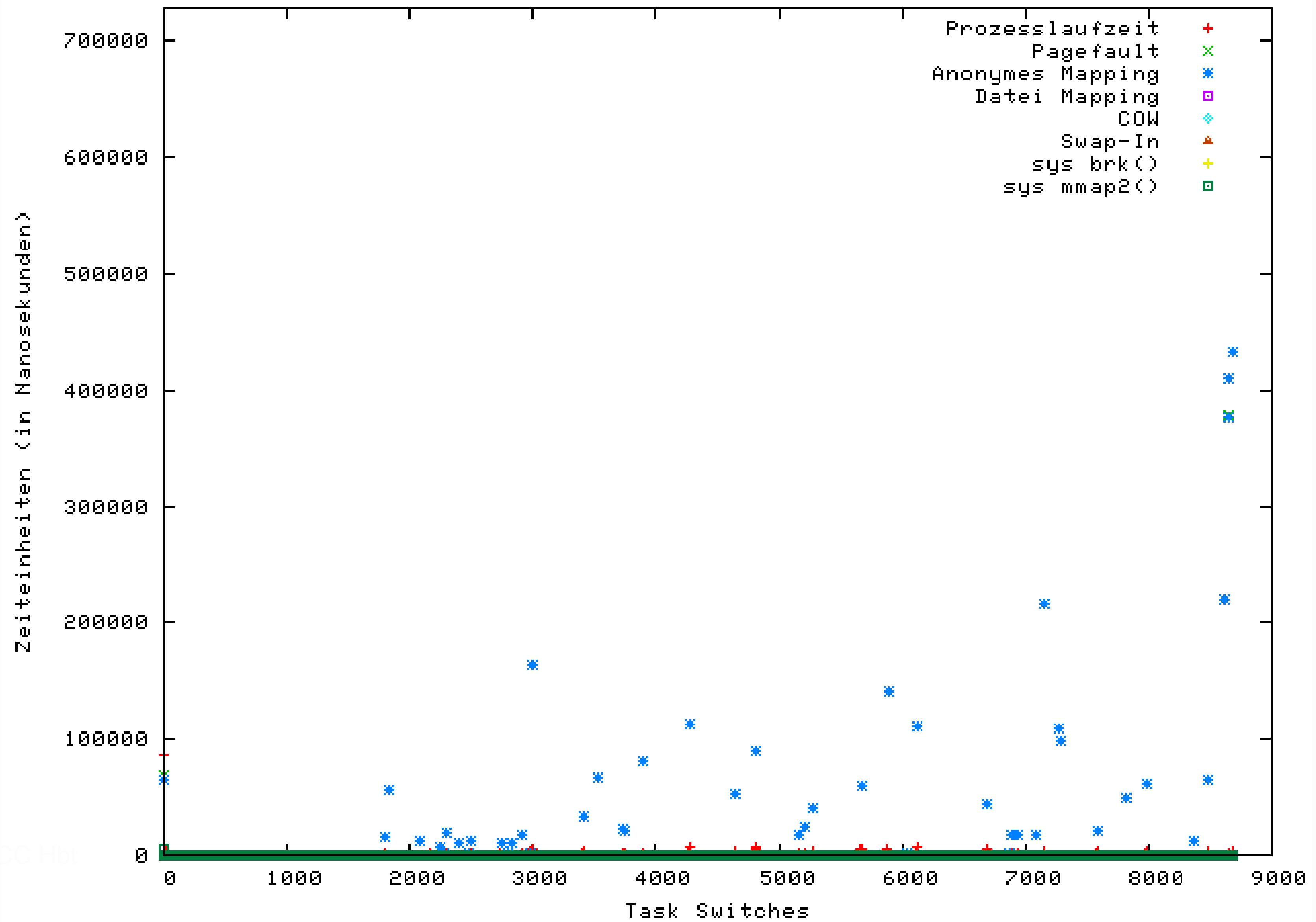


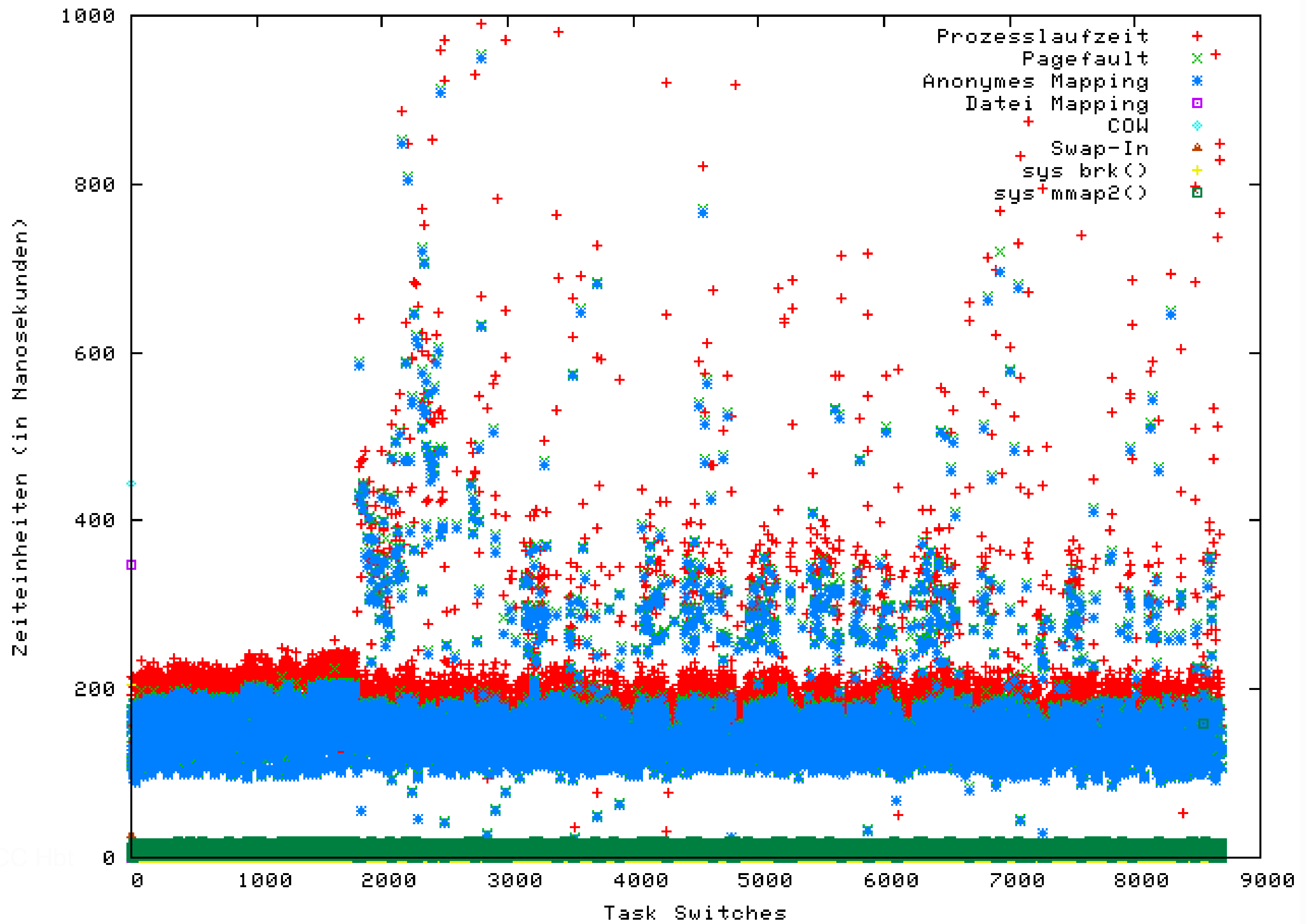
0 ns
100 ns
200 ns
brk(0)
brk(new)
PG Help
PG Anon
...

brk(2) versus mmap(2)

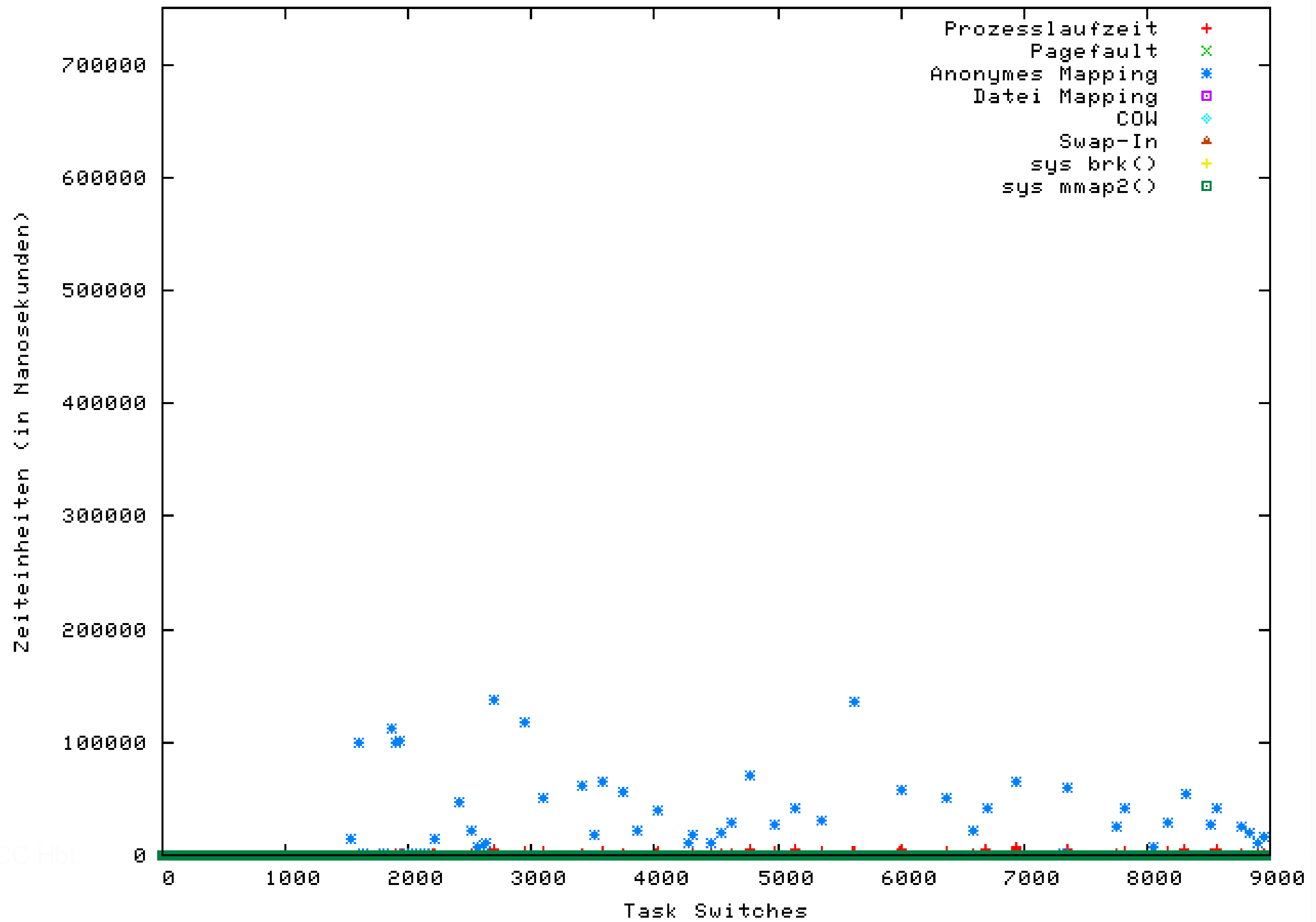


Page Fault Latenz (Allokierung von 2 * PAGE SIZE)
Runtime
Pgflt (akkumuliert)
pganon
apgno
apgfile
apgswap
apgwp
Zeiteinheiten (in 0.6GHz)
Task Switches
0
2000
4000
6000
8000
10000
12000
14000
0
2
4
6
8
10
12
14

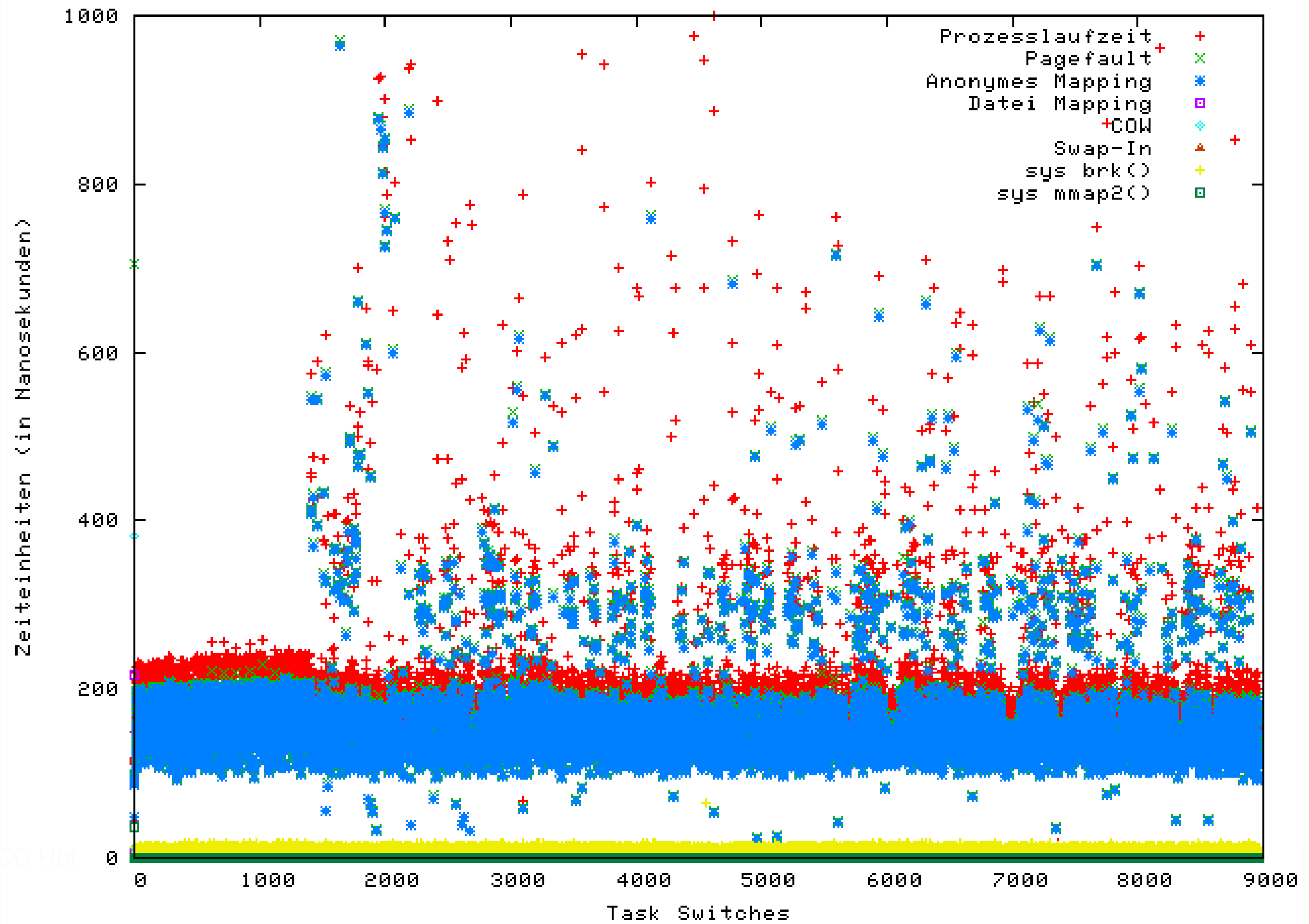

Allokierung via mmap() (bis oomkiller)
Prozesslaufzeit
Pagefault
Anonymes Mapping
Datei Mapping
COW
Swap-In
sys brk()
sys mmap2()
Zeiteinheiten (in Nanosekunden)
Task Switches
0
100000
200000
300000
400000
500000
600000
700000
1000
2000
3000
4000
5000
6000
7000
8000
9000



Allokierung via mmap() (bis oomkiller)
Zeiteinheiten (in Nanosekunden)
Task Switches
0
200
400
600
800
1000
1000
2000
3000
4000
5000
6000
7000
8000
9000
Prozesslaufzeit
Pagefault
Anonymes Mapping
Datei Mapping
COW
Swap-In
sys brk()
sys mmap2()



Allokierung via brk() (bis oomkiller)
Prozesslaufzeit
Pagefault
Anonymes Mapping
Datei Mapping
COW
Swap-In
sys brk()
sys mmap2()
Zeiteinheiten (in Nanosekunden)
700000
600000
500000
400000
300000
200000
100000
0
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
Task Switches


Allokierung via brk() (bis oomkiller)
Prozesslaufzeit
Pagefault
Anonymes Mapping
Datei Mapping
COW
Swap-In
sys brk()
sys mmap2()
Zeiteinheiten (in Nanosekunden)
Task Switches
1000
800
600
400
200
0
0
1000
2000
3000
4000
5000
6000
7000
8000
9000

Speicher Add-Ons

Java:

- Verwaltet pool aus kleinen, mittleren und grossen Blöcken
- Fallback auf malloc(3)
- By the way:

```
strace java -jar JAP.jar 2>&1 | egrep \
'((.*mmap.*MAP_ANONYMOUS)|(.*brk.*))' - | wc -l
154
```

(ja, auch Java[TM] benötigt Speicher! ;-)(`ldd java`)

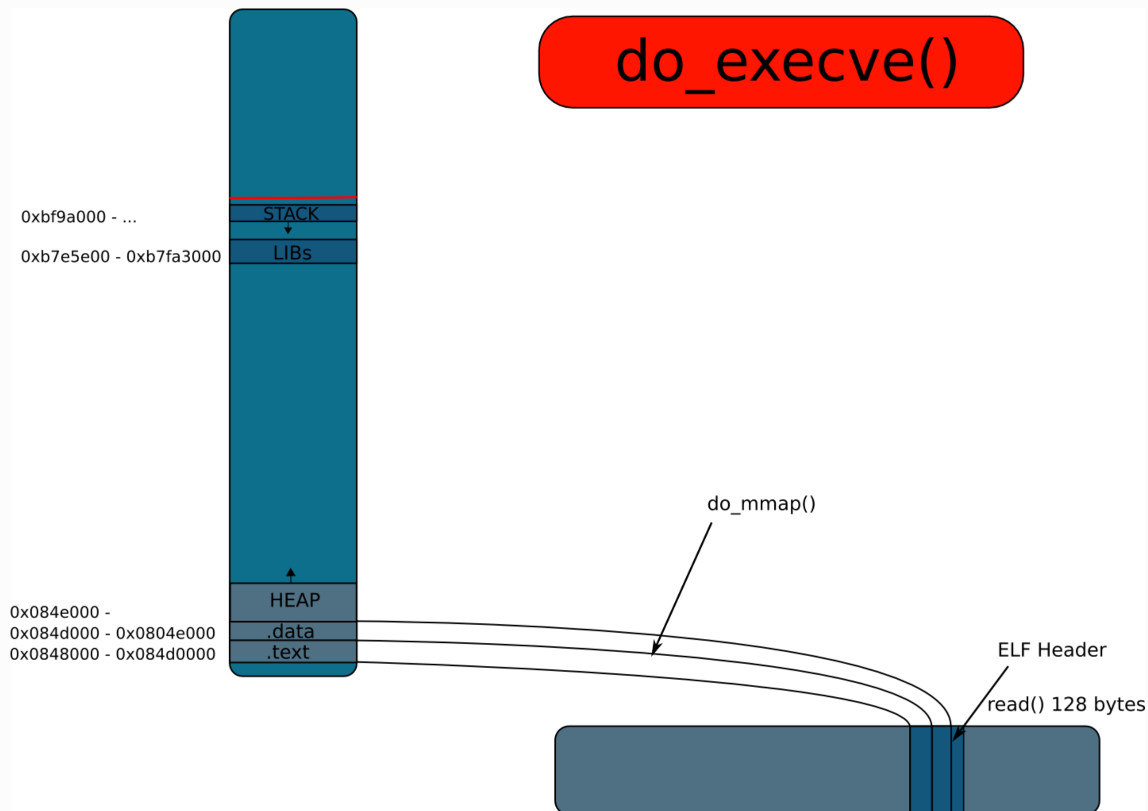
```
...
cat /proc/$(pidof java)/maps | wc -l
```

245

...und mapped eine Menge Bibliotheken! ;-)

- Ansonsten: JRE Quellcode ultra unleserlich (widerlichster C++ Code `*wuergh*`)

execve() - Detailliert



execve() - Detailliert

- Datei öffnen (`fs/exec.c`
- CPU Balancing (kleinster Speicher und Cache Penalty)
- MMU Kontext kopieren (nicht Intel; aber ia64 z.B. Feature (`asm-ia64/mmu_context.h`)
- `fs/exec.c:prepare_binprm()`
  1. Set-uid (`mode & S_ISUID` $\rightarrow$ `bprm->e_uid=inode->i_uid;`)
  2. Zugriffsberechtigungen, 128 byte read()

- `fs/exec.c:search_binary_handler()`
  - elf
  - aout
  - misc
  - script
- `fs/binfmt_elf.c:load_elf_binary()`
  1. Checks (ist es wirklich ELF)
  2. Program Header einlesen
  3. Iteration über Program Header (nach Interpreter suchen)
  4. Datei Information verwerfen (für `current`)
  5. Segmente in Speicher mappen (`elf_map()`)(mit passenden Rechten)
  6. Wenn Interpreter → `load_elf_interp()` (also mappen)
  7. Register setzen → u.a. %esp (`include/asm-i386/processor.h:sta`

8. wie immer: Aussagen verkürzt! ’-)

madvise() – paging I/O Behandlung

```
long
sys_madvise(unsigned long start, size_t len_in, int behavi
```

`behaviour` beschreibt die zu erwartende Nutzung des Speichers
Linux 2.6.15: Unterscheidet 5 Arten:

1. `MADV_NORMAL` – Standard-Verhalten
2. `MADV_SEQUENTIAL` – Sequentielles lesen
3. `MADV_RANDOM` – Zugriffe erfolgen Wahlfrei
4. `MADV_WILLNEED` – Baldigen Zugriff erwarten
5. `MADV_DONTNEED` – Keine baldigen Zugriffe erwarten

madvise_vma()

Kernel-Interne Funktion, übernimmt ”Verteilung”

```
switch (behavior) {
  case MADV_NORMAL:
  case MADV_SEQUENTIAL:
  case MADV_RANDOM:
    error = madvise_behavior(vma, prev, start, end, behav
    break;
  case MADV_WILLNEED:
    error = madvise_willneed(vma, prev, start, end);
    break;
  case MADV_DONTNEED:
    error = madvise_dontneed(vma, prev, start, end);
    break;
```

Baldiger Zugriff

`MADV_WILLNEED` veranlasst eine page-cache readahead-Operation:

```
force_page_cache_readahead(file->f_mapping,
   file, start, max_sane_readahead(end - start));
```

`mm/readahead.c:__do_page_cache_readahead()`:
Alloziert Speicher für die einzulagernden Seiten, `read_pages()` führt Leseoperation durch: `mapping->a_ops->readpage()`.

Baldiger Zugriff (cont.)

`include/linux/fs.h` definiert

```
struct  address_space_operations {
        int (*writepage)(struct page *page,
                         struct writeback_control *wbc);
        int (*readpage)(struct file *, struct page *);
        [..]
}
```

`MADV_DONTNEED` veranlasst einen Aufruf der `zap_page_range()`-Funktion.

madvise_behaviour()

- Für `MADV_NORMAL`, `_SEQUENTIAL`, `_RANDOM`:
- Kern versucht vma- Strukturen zu optimieren
- `_SEQUENTIAL` bzw. `_RANDOM` setzen `VM_SEQ_READ`bzw. `VM_RAND_READ`
- `split_vma` / `vma_merge()` übernehmen Arbeit

OOM-Killer

`select_bad_process()`: Soll einen geeigneten Prozess zum terminieren auswählen

- Durchläuft Prozessliste für `p->pid > 1`
- Berechnet via `badness()` für jeden Prozess einen Wert
- Durchlauf vorzeitig beendet, wenn:
  - Prozess im Begriff ist Speicher freizugeben (Fehler liefern)
  - Prozess hat `PF_SWAPOFF` gesetzt (wird sofort gewählt)
- `task_struct` mit höchstem Wert wird zurückgeliefert

OOM-Kill: badness

- Berechnungsgrundlage: Speicherverbrauch
- vmsize von Kindprozessen wird berücksichtigt
- `badness` $/\sqrt{cputime}$
- `badness` $/\sqrt{\sqrt{run_time}}$
- Nicewert $> 0 \rightarrow$ Wert verdoppeln
- Punkte vierteln, Falls Rawio oder euid = 0
- Um `oom_adj` Stellen shiften (Normal 0)

OOM-Kill: Weiterer Ablauf

- Zuerst wird ein Kindprozess des gewählten Prozesses gewählt
- Prozess erhält `SIGKILL`(`__oom_kill_task()`)
- `TIF_MEMDIE` wird gesetzt
- Priorität: `p->time_slice = HZ;`
- Falls kein Prozess gefunden wird:

```
panic("Out of memory and no killable processes...\n");
```

Abstraktion von Filedeskriptoren

- Intern: `struct file`
  - Enthält `struct file_operations` , diese enthält Funktionszeiger auf konkrete Implementierung (`read()`, `write()` etc.)
- Prozess besitzt `struct files_struct`
  - Verzeichnet alle von Prozess geöffneten Deskriptoren
  - Zusätzliche Daten (`FD_CLOEXEC`, spinlock, ...)
- `fs/fs_table.c:fget(unsigned int fd)` liefert entsprechende Struktur

sys_accept()

Userspace: `int clientfd = accept(listenfd, &saddr, &len)`

- `net/net/socket.c:sys_accept()`
- Zuerst: Ist `listenfd` ein Socket?
  - `fget(listenfd)`
  - `if (file->f_op == &socket_file_ops)`
- `newsock=sock_alloc();`
- `sock->ops->accept(sock, newsock, sock->file->f_flags);`
- "`return sock_map_fd()`" (`get_unused_fd()`, `filep`, `...`)

inet_accept()

`inet/ipv4/af_inet.c:inet_accept()`: Abstrahiert `AF_INET`
Etablierte, aber noch nicht im Userspace bekannte TCP-Verbindungen stehen in FIFO-artiger Queue

```
net/ipv4/inet_connection_sock.c:
struct sock *
inet_csk_accept(struct sock *sk, int flags, int *err)
```

... kontrolliert ob queue leer ist:

- Nein: älteste `sock`-Struktur aus Queue entfernen
- Ja: `inet_csk_wait_for_connect()` → `TASK_INTERRUPTIBLE`

Links:

Kernel Code Browser: `http://lxr.linux.no/`

Das Allerletzte / Linux Intern

```
$ cd /usr/src/linux &&
$ egrep -ri \
 '(fixme|xxx|crap|junk|shit|fuck|broken|b0rken)' * \
 | wc -l
23075
```

FIN

- Fragen/Anmerkungen/Pizza/Bier
- ```
  if(current == rq->idle) {
   /* Hagen Paul Pfeifer <hagen@jauu.net> */
   /*   KeyID: 0x98350C22                 */
   /* Florian Westphal <fw@strlen.de>     */
   /*   KeyID: 0xF260502D */
   panic("additional information needed");*/
  }
  ```
  ```