

# Netzwerk-Programmiermodelle und I/O Optimierungen für Unix Betriebssysteme

Hagen Paul Pfeifer • Florian Westphal  
[hagen@jauu.net](mailto:hagen@jauu.net) [fw@strlen.de](mailto:fw@strlen.de)

<http://www.protocol-laboratories.net>

14. Oktober 2006

# Vortragsfahrplan

## 1. Programmiermodelle

- ▶ Asynchrone I/O, fork(), threads, select(), poll() /dev/poll, epoll, kqueue, kevent

## 2. I/O Optimierungen

- ▶ Zero-Copy, Read-Write, Mmap, Sendfile, Splice und Tee

## 3. Kernelspace Optimierungen

- ▶ NAPI
- ▶ Dynamic Right Sizing

# Kapitel 1

# Programmiermodelle

# Server/Client Programmiermodelle

- ▶ `fork()`: Ein Prozess pro Client. Vorteile:
  - Einfache Programmierung (z.B.: Fehlerhandling mit `_exit()`)
  - Gesamt-System weniger Fehleranfällig
- ▶ Design ist von Nachteil, wenn:
  - ... Prozesse untereinander kommunizieren müssen
  - ... viele Clients ( $> 1000$ ) zu erwarten sind
- ▶ Overhead durch `fork()`, `_exit()`, `waitpid()`
- ▶ Performance sehr vom Scheduler abhängig (Linux 2.4:  $O(n)$ , 2.6:  $O(1)$ )

# Server/Client Programmiermodelle

- ▶ Ein Thread pro Client. Vorteile:
  - Threaderzeugung meist schneller als `fork()` (Linux 2.6: ca.  $\frac{1}{16}$  CPU Zyklen)
  - Programmiermodell dem zu `fork` sehr ähnlich
- ▶ Nachteile:
  - Bei gemeinsam genutzten Daten ist Synchronisation notwendig
- ▶ Performance von Scheduler und Thread-Modell ( $m : n, 1 : 1$ ) abhängig

# Server/Client Programmiermodelle

- ▶ Multiplexer: Ein Prozess/Thread für mehrere (oder alle) Clients. Zwei Möglichkeiten:
  1. Level-Triggered: ("Readiness Notification")
    - fd wird gemeldet, sobald er bereit ist: `select`, `poll`, Async I/O, etc.
  2. Edge-Triggered: ("Change Notification")
    - fd wird nur gemeldet, wenn eine Zustandsänderung eintritt  
z.B. Linux RT-Signals; Optional: `epoll`, `kqueue`, Async I/O

# Asynchrone I/O

## ▶ Vorgehen:

- `struct aiocb` mit Deskriptor+buffer+Länge
- `aio_read`, `aio_write` etc. starten Schreib/Lesevorgang
- I/O läuft "im Hintergrund" ab, Mitteilung wann beendet entweder mittels Signal oder Polling (`aio_suspend`)

Nachteil: Auch `open()` kann blockieren; `aio_open` nicht existent

Alternative: Worker Threads

# Multiplexing-Mechanismen: select

## ▶ "Der Klassiker:" 4.2 BSD (1983)

```
int select(int nfd, fd_set *readfds, fd_set *writefds,  
           fd_set *exceptfds, struct timeval *timeout);
```

- Gewöhnungsbedürftiges API (nfd: Anzahl zu untersuchender Bits)
  - nur Deskriptoren kleiner FD\_SETSIZE erlaubt
  - manche Systeme (Linux) erfordern O\_NONBLOCK für alle fd
- ▶ Am weitesten Verbreitet (Sogar unter Winsock verfügbar ...)



# Multiplexing-Mechanismen: poll

## ▶ SVR3 (1986)

```
struct pollfd {  
    int    fd;  
    short events;  
    short revents;  
};
```

```
int poll(struct pollfd p[], nfds_t nfds, int timeout);
```

- manche Systeme (Linux) erfordern `O_NONBLOCK` für alle `fd`
- Array bei vielen Veränderungen der Größe ungünstig
- `pollfd[]` muss bei jedem Syscall zweifach kopiert werden

# Probleme

## ▶ Probleme:

- "Doppelte Arbeit": Applikation und Kernel verwalten dieselben Informationen
- Lineare Suche im `pollfds` Array (`poll`)
- Man erhält stets auch alle "ereignislosen" Deskriptoren

## ▶ Wünschenswert:

- Man erhält nur noch Deskriptoren, die für die gewünschte Operation zu Verfügung stehen

# SunOS: /dev/poll

## ▶ /dev/poll:

- setzt auf `poll()` auf, neue Gerätedatei `/dev/poll`
- Behandlung mit "alten" Syscalls: `open`, `read`, `write`

## ▶ Vorgehen:

- `int fd = open("/dev/poll", O_RDWR);`
- `pollfd` Struktur(en) initialisieren und nach `fd` schreiben
- via `ioctl()` auf Ereignisse warten
- Relevante `pollfd` Strukturen werden in Userspace Buffer kopiert

# Linux 2.6: epoll

- ▶ Erster Patch: Linux 2.5.44 (2002)
- ▶ Neues API: 3 neue Syscalls

```
struct epoll_event{uint32_t events;  
                    epoll_data_t data;}
```

- ▶ Level oder (optional) Edge Triggered
- ▶ letztes `close()` auf fd entfernt diesen Automatisch

# epoll: Vorgehen

- ▶ `int efd = epoll_create(hint);`
- ▶ `epoll_event` initialisieren;  
z.B `epoll_event e = { .events = EPOLLIN };`
- ▶ hinzufügen: `epoll_ctl(efd, EPOLL_CTL_ADD, fd, &e);`
- ▶ Mit `epoll_wait` auf Ereignisse warten
- ▶ Edge-Triggered Verhalten kann pro fd eingeschaltet werden: `e.events |= EPOLLET;`

## \*BSD: kqueue

- ▶ FreeBSD 4.1 (2000), OpenBSD 3.5 (2004), NetBSD 2.0 (Dez. 2004)

```
struct kevent {
    uintptr_t ident; /* identifier for this event */
    short     filter; /* filter for event */
    u_short   flags; /* action flags for kqueue */
    u_int     fflags; /* filter flag value */
    intptr_t  data; /* filter data value */
    void      *udata; /* opaque user data identifier */
};
```

- ▶ 2 Syscalls, ein Makro
- ▶ Filterkonzept: u.a. für Timer und Signale

# Vorgehen

- ▶ `int kfd = kqueue()`
- ▶ `struct kevent` initialisieren, z.B.  
`EV_SET(&kev, fd, EV-`  
`FILT_READ, EV_ADD|EV_ENABLE, 0, 0, 0);`
- ▶ `kevent()` aktualisiert `fd` set und/oder wartet auf Ereignisse

# Kevent

- ▶ Linux hat kein einheitlichen Mechanismus auf Events zu reagieren
- ▶ Ziel: möglichst viele Event so schnell wie möglich zu bearbeiten mit wenig System-Overhead
- ▶ Probleme bei asynchronen I/O
- ▶ Evgeniy Polyakov Patch - kevent
- ▶ Kombination und Ideen aus AIO/kqueue Interface
- ▶ Ringpuffer welcher `gemmap()` ed wird
- ▶ kvent kann auf folgende Ereignisse reagieren:
  - Alles was `poll()` abdeckt `KEVENT_POLL`
  - Neue Netzwerk-Daten oder Verbindungen `KEVENT_SOCKET`



- `inotify()` Ereignisse `KEVENT_INODE`
  - Netzwerk asynchrone I/O Ereignisse `KEVENT_NAIO`
  - Timer Ereignisse `KEVENT_TIMER`
- ▶ Erste Zahlen (httpperf, 40K Verbindungen):
- `kevent`: 3388.4 req/s
  - `epoll` und `kevent_poll` 2200-2500 req/s

# Zusammenfassung

- ▶ Alternativen gegeneinander abwägen
  - Protokoll, Zugriffsmuster
  - Betriebssystem(e), Portabilität
- ▶ `select`: portabelste Schnittstelle, aber Problematisch bei vielen Deskriptoren
- ▶ `poll` : Array; lineare Suche; Array muss verwaltet werden
- ▶ Alternativen sind Systemspezifisch (Wrapperbibliotheken wie `libevent` existieren)
- ▶ KISS – `fork()` kann durchaus Mittel der Wahl sein!

# Kapitel 2

## I/O Optimierungen

# Zero-Copy

## ▶ Typische Lehrbuch-Serveranwendung:

```
while((cnt = read(filefd, buf, buflen)) > 0) {
    char *bufptr = buf;
    do {
        ssize_t written = write(socket, bufptr, cnt);
        /* handle errors, if any ... */
        cnt -= written;
        bufptr += written;
    } while (cnt > 0);
}
```

# Zero-Copy

- ▶ Positiv: einfach und portabel
- ▶ Negativ: nicht wirklich Performant
  1. Kopieraktion hohe Kosten
  2. Syscalls verursachen weiteren Overhead
  3. Kontext-Wechsel:
    - Register sichern
    - Cache konsistent mit VM-Mappings, bei neuen VM-Mapping -> TLB flushen, (invXpg)
    - Kein neues VM-Mapping: Kernel-Thread oder US-Threads
    - Einige Architekturen müssen auch Cache flushen (x86: NULL-Makro)

- Cold Caches

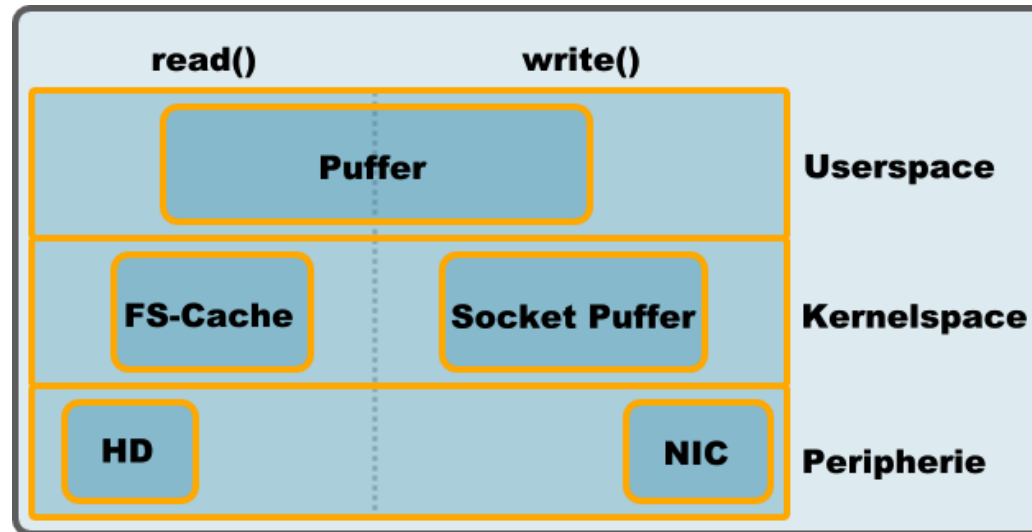
## 4. Information Bonbon:

(microseconds, lmbench, v2.6.10)

cpu	clock speed	context switch
pentium-M	1.8GHz	0.890
dual-Xeon	2GHz	7.430
Xscale	700MHz	108.2
dual 970FX	1.8GHz	5.850
ppc 7447	1GHz	1.720

# Read() - Write()

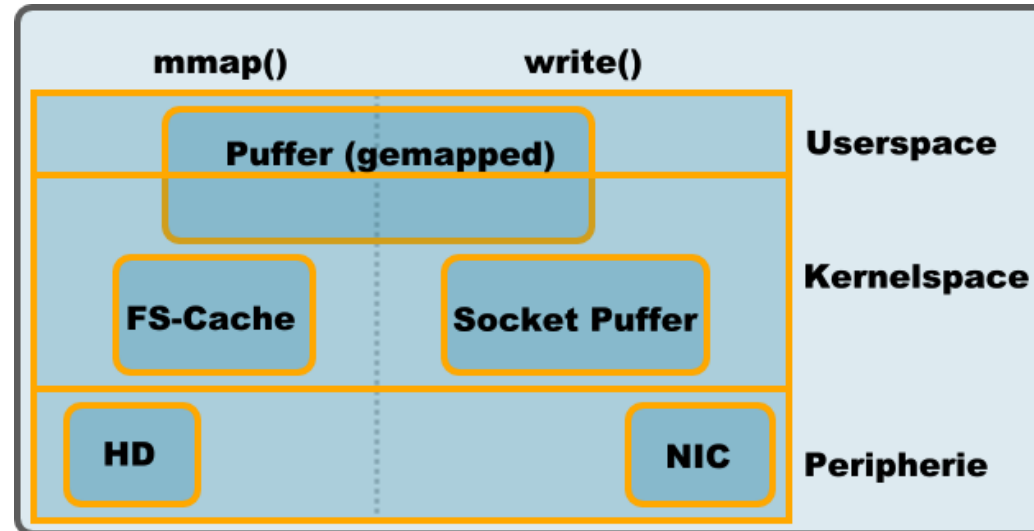
- ▶ Read-Write näher betrachtet:



- ▶ Speicher muss vier mal kopiert werden (nicht nötig)
- ▶ Vier Kontext-Switches
- ▶ Von Festplatte und nach NIC normalerweise via DMA

# Mmap()

- ▶ Mmap-Write näher betrachtet:

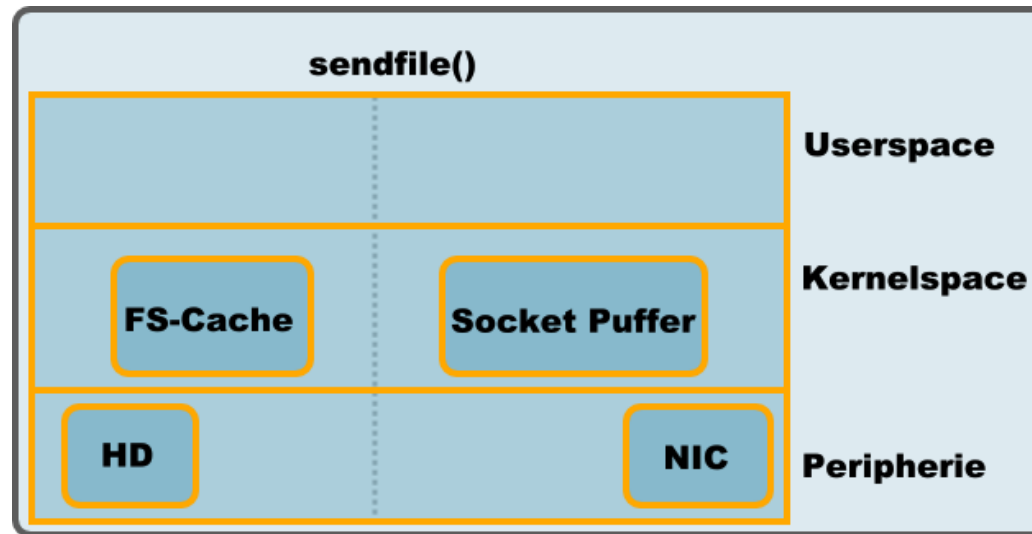


- ▶ Auch vier Kontext-Switches
- ▶ Aber: Kernelpuffer wird geteilt mit Userspace
- ▶ Zwei mal DMA Transfer, Ein mal CPU-Transfer
- ▶ Problem: anderer Prozeß stutzt Datei (SIGBUS, RT\_SIGNAL\_LEASE)



# Sendfile

- ▶ `ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count);`



- ▶ Ein Kontextwechsel
- ▶ Zwei mal DMA Transfer, Ein mal CPU-Transfer
- ▶ Nicht portabel (Prototyp unterscheidet sich)
- ▶ Wenige Anwendungen nutzen `sendfile`

(man munkelt Linux hat sendfile nur Aufgrund von Druck der Apache-Gemeinde ;-)

# Sendfile (Scatter/Gather IO)

- ▶ Wie vanilla sendfile, aber CPU-Kopie entfällt
- ▶ Kein kontinuierlicher Speicherbereich benötigt (Header, Payload)
- ▶ `sk_buff` enthält Zeiger auf Datenbereich (FS-Cache) (Analogie: `writenv`)
- ▶ `dev->feature |= NETIF_F_SG` (`linux/netdevice.h`)
- ▶ Vorteile:
  - komplette Kopiervorgang kann eingespart werden
  - \*-Cache wird nicht „verunreinigt“
- ▶ 3Com (3CR990 Family): 16 SG-Einträge

# Sendfile unter Solaris/BSD/Windows

- ▶ Solaris
  - `sendfile()` und `sendfilev()`
- ▶ Microsoft Windows
  - `TransferFile()`

# Abstecher: madvise

- ▶ `int madvise(void *start, size_t length, int advice);`
- ▶ Gibt BS Tipp wie Speicherbereich gelesen/verwendet wird
- ▶ `MADV_RANDOM`, `MADV_SEQUENTIAL`, `MADV_DONTNEED`
- ▶ `fadvise()` Pendant für Dateien
- ▶ `readahead()`

# Sendfile und TCP\_CORK

- ▶ Oft: generierter Header und Datei als Daten (HTTP)
- ▶ Zwei `write()` generieren zwei Pakete (oft)

```
setsockopt(fd, IPPROTO_TCP, TCP_CORK, &on, sizeof(on));  
write(fd, http_header, strlen(header));  
sendfile(fd, file_fd, &offset, nbytes);  
setsockopt(sock, IPPROTO_TCP, TCP_CORK, &off, sizeof(off));
```

- ▶ TCP\_CORK off: Flushed Puffer sofort
- ▶ BSD, OS X: TCP\_NOPUSH
- ▶ Portabler: (und nur ein Systemaufruf)

```
struct iovec { void *iov_base; size_t iov_len; };  
ssize_t writev(int fd, const struct iovec *vector,  
               int count);
```

# Splice und Tee

## ▶ Splice

- `long splice(int, loff_t, int, loff_t, size_t, unsigned int);`
- Kernelspeicher im KS für US (Userpipe im KS)
- Userspace hat Kontrolle über diesem Bereich
- Bewegt Daten von/nach dem Puffer nach/von Deskriptor
- Im-Kernel Ersatz für `read/write` zum Puffer
- Nur wenn keine Bearbeitung am Daten
- Steigerung der Datentransferrate um bis 70% und 50% CPU
- Asynchron: SIGIO via `fcntl(2)`

- Neu in 2.6.17-rc1
- Anwendungen:
  - Socket zu Socket (redirect)
  - Datei nach Datei

## ▶ Tee

- `long tee(int, int, size_t, unsigned int);`
- Kopiert Daten im KS-Puffer
- `memcpy()` von Kernelpuffer zu Kernelpuffer
- Anwendungen:
  - MPEG-Stream nach Datei **und** Socket
- `tee(1)` Pseudo-Implementierung

```
while(1) {  
    tee(STDIN_FILENO, STDOUT_FILENO, ...);  
}
```



```
splice(STDIN_FILENO, filefd, ...)  
}
```

# Kapitel 3

## Kernelspace Optimierungen

# NAPI

- ▶ Ansatz um Netzwerkperformance Performance zu erhöhen
- ▶ Problem: Overhead bei Interrupts
- ▶ Ansatz:
  - Interrupt startet Verarbeitungsroutine
  - Interrupts werden gesperrt (`Softirq`)
  - `dev->poll()` verarbeitet alle Pakete im Ring-Puffer
- ▶ Weniger Interrupts, weniger Unterbrechungen
- ▶ 1Gbit NIC:  $12 \mu s$
- ▶ Aber: moderne NIC's Interrupts Moderation
- ▶ Tipp: e1000 Implementierung

# Dynamic Right Sizing

- ▶ Empfangspuffer konservative Größe (Embedded Geräte)
- ▶ Anpassungen des Empfang- und Sendepuffer
- ▶ Problem bei LFN (Bandbreiten-Verzögerungs Produkt)
- ▶ Weg von manuellen anpassen der Puffer (`setsockopt()`)
- ▶ Ziel:
  - Netzwerk-Pipe voll
  - Möglichst wenig Kernelspeicher (Problem: 10k Verbindungen)

# Tuninig Tipps

## ► Für die Netzhacker mit GE

```
ifconfig eth0 mtu 9000
sysctl -w net.ipv4.tcp_sack=0
sysctl -w net.ipv4.tcp_timestamps=0
sysctl -w net.core.rmem_max=524287
sysctl -w net.core.wmem_max=524287
sysctl -w net.core.optmem_max=524287
sysctl -w net.core.netdev_max_backlog=300000
sysctl -w net.ipv4.tcp_rmem="10000000 10000000 10000000"
sysctl -w net.ipv4.tcp_wmem="10000000 10000000 10000000"
sysctl -w net.ipv4.tcp_mem="10000000 10000000 10000000"
sysctl -w net.ipv4.tcp_tw_recycle=1
sysctl -w net.ipv4.tcp_tw_reuse=1
```

# Kapitel 4

## Prolog

# Optimierungs Baustellen

- ▶ TCP Metrics (Linux)
- ▶ Plugable TCP Congestion Control
- ▶ TOE - TCP Offload Engine
- ▶ TCP Segmentation Offloading (TSO)
- ▶ Net Channels - Van Jacobson
- ▶ FireEngine (Solaris)

# Weitererführende Links, Quellen

- ▶ `man select poll epoll kqueue ...`
- ▶ Dan Kegel's "The C10K problem"  
<http://www.kegel.com/c10k.html>
- ▶ `libevent`, <http://www.monkey.org/~provos/libevent/>
- ▶ <http://people.freebsd.org/~jlemon/papers/kqueue.pdf>
- ▶ Stevens, Rago: "Advanced Programming in the Unix Environment"
- ▶ Stevens: "UNIX Network Programming, The Sockets Networking"
- ▶ McKusick, Neville-Neil "The Design and Implementation of the FreeBSD Operating System"
- ▶ Dynamic Right Sizing:



<http://woozle.org/~mfisk/papers/tcpwindow-lacsi.pdf>

▶ **INVLPG:**

<http://www.cs.tut.fi/~siponen/upros/intel/instr/invlpg>

# FIN

- ▶ Danke für eure Aufmerksamkeit!
- ▶ Fragen – Anregungen – Bemerkungen?
- ▶ EMail: [hagen@jauu.net](mailto:hagen@jauu.net)
  - Key-ID: 0x98350C22
  - Fingerprint: 490F 557B 6C48 6D7E 5706 2EA2 4A22 8D45 9835 0C22
- ▶ EMail: [fw@strlen.de](mailto:fw@strlen.de)
  - Key-ID: 0xF260502D
  - Fingerprint: 1C81 1AD5 EA8F 3047 7555 E8EE 5E2F DA6C F260 502D