Skalierbare High-Performance Netzwerkprogrammierung unter Linux

Felix von Leitner <felix-linuxtag@codeblau.de>

Hochperformanter, gut skalierender Netzwerk-Code ist heute eines der wichtigsten Verkaufsargumente für Server-Betriebssysteme. Die Standard-Interfaces von POSIX und der Open Group eignen sich nur bedingt für hohe Skalierbarkeit.

Daher haben alle Anbieter in diesem Marktsegment eigene Interfaces entwickelt. Dieser Artikel beleuchtet die Unterschiede stellvertretend an der Evolution dieser Interfaces unter Linux.

Inhaltsverzeichnis

Grundlagen der Netzwerk-Programmierung 1
Handhabung mehrerer paralleler Verbindungen 2
- Ein Prozeß pro Verbindung 2
- Skalierbares Scheduling 3
- Kontextwechsel und Prozeßerzeugungskosten 4
- Timeout-Handling 4
- Das Multithreading-Debakel 5
- Asynchroner I/O 6
- Linux 2.4: SIGIO 6
- Solaris: /dev/poll 8
- Linux: epoll 8
- Windows: Completion Ports 9
- FreeBSD: kqueue 9

Sonstige Tricks zur Performance-Steigerung 9
- writev, TCP_CORK und TCP_NOPUSH 9
- sendfile 10
- Sonstige Performance-Tweaks 10

Grundlagen der Netzwerk-Programmierung

Unter Unix werden Geräte und IPC-Mechanismen wie Dateien angesprochen. Man öffnet eine Datei mit `open()`, liest mit `read()`, schreibt mit `write()` und schließt die Datei wieder mit `close()`. Hier ist ein kurzer Beispiel-Code:

```
char buf[4096];
int fd=open("index.html", O_RDONLY);
int len=read(fd,buf,sizeof buf);
write(outfd,buf,len);
close(fd);
```

Man beachte, daß `open()` einen Integer zurückliefert, d.h. eine Zahl. Diese dient als Referenz auf ein Kernel-Objekt; man kann damit in seinem Programm nichts tun, als sie dem Kernel gegenüber als Referenz zu benutzen. Damit Rechnen ist sinnlos, und es gibt auch keine Möglichkeit, zu einem solchen File Deskriptor den Dateinamen zu erfragen.

Für jeden Prozeß fangen die Deskriptoren bei 0 an, und 0 bis 2 sind für Eingabe, Ausgabe und Fehlermeldungen reserviert. POSIX sagt, daß der Kernel als neuen Deskriptor immer den kleinsten noch nicht vergebenen zuweist. Mehrere Prozesse können (verschiedene) Deskriptoren für die selbe Datei haben.

Das API ist offensichtlich: `read()` kehrt zurück, wenn es Daten gelesen hat. `write()` kehrt zurück, wenn die Daten geschrieben wurden. Danach kann man den Puffer anderweitig verwenden.

Netzwerkprogrammierung funktioniert analog. Die Deskriptoren verweisen nicht auf Dateien, sondern auf Sockets (Steckdose). Einen Socket richtet man mit `socket()` ein. Hier ist ein Code-Beispiel:

```
char buf[4096];
int len;
int fd=socket(PF_INET,SOCK_STREAM,IPPROTO_TCP);
struct sockaddr_in si;

si.sin_family=PF_INET;
inet_aton("127.0.0.1",&si.sin_addr);
si.sin_port=htons(80);
connect(fd,(struct sockaddr*)si,sizeof si);
write(fd,"GET / HTTP/1.0\r\n\r\n");
len=read(fd,buf,sizeof buf);
close(fd);
```

Dieser Code öffnet eine TCP-Verbindung zum Rechner 127.0.0.1 auf Port 80 und fragt nach /. Es handelt sich hier also um einen simplen HTTP-Client.

Das Socket-API trennt das Erstellen des Sockets von dem Erstellen einer Verbindung. Der Hintergrund sind Server, die sich nie mit einer Gegenstelle verbinden, sondern zu denen die Gegenstellen sich verbinden. Hier ist ein Mini-Server:

```
int len,s,fd=socket(PF_INET,SOCK_STREAM,IPPROTO_TCP);
struct sockaddr_in si;

si.sin_family=PF_INET;
inet_aton("127.0.0.1",&si.sin_addr);
si.sin_port=htons(80);
bind(fd,(struct sockaddr*)si,sizeof si);
listen(fd,16);
s=accept(fd,(struct sockaddr*)&si,&len);
```

Handhabung mehrerer paralleler Verbindungen

Wie kann ein Server mehr als einen Client verwalten? Offensichtlich ist das Modell mit dem blockierenden `read()` und `write()` nicht geeignet, um in einem Prozeß mehr als eine Verbindung zu halten - man könnte immer nur Daten über eine Verbindung zur Zeit schicken. Schlimmer noch, sobald der erste Client mal nichts schickt, bleibt das `read()` und damit die anderen Clients hängen.

Die traditionelle Lösung dafür ist, pro Verbindung einen neuen Prozeß zu starten.

Ein Prozeß pro Verbindung

Unter Unix ist das glücklicherweise sehr einfach; man kann den gerade laufenden Prozeß einfach klonen, man muß nicht ein weiteres Programm dafür von der Festplatte laden, und der neue Prozeß übernimmt auch alle Variablen und offenen File Deskriptoren.

Trotzdem hat dieses Modell Nachteile und Probleme, insbesondere bei der Skalierbarkeit. Typische Betriebssysteme sind für den Fall von ein paar Dutzend bis zu Hundert Prozessen optimiert.

Dazu kommt, daß pro Prozeß normalerweise bedeutende Mengen Systemspeicher verbraucht werden, insbesondere bei dynamisch gelinkten Programmen. 10.000 Clients gleichzeitig bringen auch größere Server zum Swappen, selbst wenn die Clients gar keine Daten anfordern. Manche Systeme (Solaris, Windows) brauchen außerdem sehr lange für die Prozeßerzeugung und eignen sich daher für dieses Modell nicht.

Auch stellt sich bei diesem Modell die Frage, wie man Timeouts realisieren soll. Wenn ein Client nach dem Verbindungsaufbau für 20 Minuten nichts sagt, sollte der Server in der Lage sein, das zu erkennen und die Verbindung zu schließen.

Skalierbares Scheduling

Das Betriebssystem verwaltet eine Liste von Prozessen und unterbricht periodisch (üblicher Wert: 100 Mal pro Sekunde) den laufenden Prozeß, um andere laufen zu lassen. Dieser Teil des Systems, der den neuen Prozeß auswählt, heißt Scheduler.

Wenn ein Prozeß wegen eines `read()` oder `write()` blockiert, wählt der Scheduler auch einen neuen Prozeß. Wenn also hunderte von Prozessen konkurrierend lesen und schreiben und dabei ständig blocken, dann verbringt das System einen signifikanten Teil der CPU-Zeit im Scheduler.

Typischerweise laufen auf einem System eine paar Dutzend Prozesse, von denen ein oder zwei tatsächlich laufen wollen, während die anderen gerade auf eine Eingabe oder ein Signal warten. Es ist also sinnvoll, wenn das System zwei Prozeßlisten hat - eine für die laufen wollenden Prozesse und eine für die auf Eingabe wartenden. Die erste Liste wird dann sehr kurz sein und der Scheduler spart sich das Anschauen der großen Gesamtliste.

Unix hat einen Mechanismus, um interaktive Prozesse zu bevorzugen und Batch-Prozesse zu benachteiligen. Dazu wird sekündlich für alle Prozesse der `nice`-Wert neu berechnet. Dafür schaut sich der Scheduler alle Prozesse an, und muß dabei bei einer sehr großen Prozeßanzahl sehr viele Daten lesen. Das führt besonders bei älteren oder Desktop-CPUs zum Cache Trashing.

Ein weiteres typisches Problem ist auch, daß die Prozeßtabelle mit einem Kernel Lock geschützt wird. Bei einem System mit mehr als einem Prozessor wird also kann also der zweite Prozessor während der Nice-Berechnung keinen Prozeßwechsel vornehmen. Dieses Problem lösen kommerzielle Unix-Varianten gewöhnlich, indem sie die Datenstruktur auf die Prozessoren verteilen, so daß jeder Prozessor eine eigene Run Queue hat.

Es gibt noch weiteren Spielraum für Optimierungen im Scheduler. Man kann z.B. die Run Queue sortiert vorhalten, und hierfür bieten sich diverse schlaue Datenstrukturen wie Heaps an, die den Aufwand von O(n) auf O(log n)[1] senken können.

Der heilige Gral der Scheduler ist allerdings der O(1)[2]-Scheduler. Im Gesamtsystem wird die Leistung nie völlig unabhängig von der Anzahl der Prozesse sein können, weil die Leistung heutiger Prozessoren stark davon abhängig ist, daß alle häufig benutzten Daten im Prozessor-Cache Platz finden.

Der Linux 2.4 Scheduler hat eine einzelne unsortierte Run Queue mit allen lauffähigen Prozessen ("runnable"), und eine Liste mit allen blockenden Prozessen und Zombies. Alle Operationen auf der Run Queue sind hinter einem Spinlock geschützt. Der Scheduler bemüht sich um CPU-Affinität[3]

Für Linux gibt es diverse neue Scheduler, mit Heaps und multiplen Run Queues, aber der erstaunlichste Scheduler ist der O(1)-Scheduler von Ingo Molnar. Dieser Scheduler ist in den 2.5 "Hacker-Kerneln" Standard.

Der O(1)-Scheduler hat pro CPU zwei Arrays mit jeweils einer verketteten Liste. Pro möglicher Prozeß-Priorität gibt es eine Liste in diesen Arrays. Da alle Einträge in diesen Listen die gleiche Priorität haben, müssen die Listen nicht weiter sortiert werden. Das eine Array pro CPU hat die lauffähigen Prozesse, das andere ist leer. Das Ein- und Austragen eines Prozesses in einer doppelt verkettete Liste ist O(1). Der Scheduler nimmt sich also immer die volle Liste, läßt jeden Prozeß darin einmal laufen, trägt den Prozeß dann in dieser Liste aus und in der anderen Liste ein, und wenn die Liste leer ist, tauscht er die Zeiger auf die Arrays aus. Die `nice` Berechnung wird

[1] Vereinfacht gesagt heißt O(n), daß der Aufwand linear mit der Größe der Datenstruktur wächst (10.000 Prozesse: Aufwand 10.000), während bei O(log n) der Aufwand nur logarithmisch steigt (10.000 Prozesse: Aufwand 14).
[2] O(1) zeigt an, daß der Aufwand für den Scheduler konstant ist, unabhängig von der Anzahl der Prozesse im System
[3] Damit ist gemeint, daß bei einem SMP-System ein Prozeß nicht ständig von einem Prozessor zum nächsten wechselt, weil jeder Prozessor seinen eigenen Cache mitbringt, der dann nicht optimal ausgenutzt wird.

in diesem Scheduler umgekehrt: jetzt werden nicht mehr interaktive Prozesse belohnt, sondern Batch-Prozesse werden bestraft (interaktive Prozesse blocken, Batch-Prozesse werden vom Scheduler unterbrochen).

Dieser geniale Scheduler skaliert praktisch unabhängig von der Anzahl der Prozesse. In Tests hat man selbst bei 100.000 gestarteten Threads keine Verlangsamung beobachtet.

Kontextwechsel und Prozeßerzeugungskosten

Der Kontextwechsel ist auf heutigen Prozessoren eine der langsamsten Operationen überhaupt. Ein Kontextwechsel findet statt, wenn ein Prozeß zu einem anderen gewechselt wird (der Übergang von einem Prozeß zum Betriebssystem ist auch sehr aufwendig, aber dafür gibt es häufig sehr effizienten Hardware-Support).

Bei einem Kontextwechsel muß das Betriebssystem die Register sichern und laden, den TLB flushen und bei manchen Architekturen muß auch manuell die Pipeline geleert werden und die Caches geflusht werden. Auf einem üblichen Desktop-System mit laufendem mp3-Player kommen 10-70 Kontextwechsel pro Sekunde vor. Wenn ein HTTP-Benchmark über Loopback läuft, sind es 10.000, über ein Ethernet-Interface sogar 20.000. Offensichtlich zählt hier jede Mikrosekunde. Architekturen mit wenig Registern wie x86 haben hier einen Erbvorteil gegenüber Architekturen mit vielen Registern oder komplizierten Anordnungen wie SPARC oder IA-64.

Der Speicherbedarf eines leeren Prozesses ist bei aktuellen Linux-System in der Tat erschreckend groß. Schuld ist aber nicht Linux, sondern die libc unter Linux, die GNU libc. Diese verbraucht mit großem Abstand von allen libc-Implementationen den meisten Speicher und hat die höchsten Startup-Kosten.

Ich habe daher eine eigene libc für Embedded-Systeme und Server-Anwendungen wie CGI-Programme und Servlets geschrieben, und damit erreicht man ein statisches Hallo-Welt-Binary von unter 300 Bytes. Das Ausführen dieses Programmes inklusive `fork()`, `exec()` und `wait` kostet auf einem Athlon ca. 200.000 CPU-Zyklen. Ein 2 GHz Athlon kann also rein rechnerisch pro Sekunde 10.000 Prozesse erzeugen.

Timeout-Handling

Die einfachste Methode zum Timeout-Handling ist `alarm()`. `alarm(10)` schickt dem aktuellen Prozeß nach 10 Sekunden ein Signal; wenn man dieses nicht explizit abfängt, wird der Prozeß dann vom System beendet. Für triviale Netzwerk-Servlets ist das wie geschaffen.

Unix bietet aber auch andere Methoden, um einen Timeout zu implementieren. Die bekannteste (und älteste - eingeführt 1983 mit 4.2BSD) ist `select()`:

```
fd_set rfd;
struct timeval tv;
FD_ZERO(&rfd); FD_SET(0,&rfd); /* fd 0 */
tv.tv_sec=5; tv.tv_usec=0;    /* 5 Sekunden */
if (select(1,&rfd,0,0,&tv)==0) /* read, write, error, timeout */
  handle_timeout();
if (FD_ISSET(0, &rfd))
  can_read_on_fd_0();
```

Das erste Argument ist die Nummer des größten übergebenen Filedeskriptors plus 1 (das ist die eine Ausnahme, bei der Rechnen mit den File Deskriptoren doch sinnvoll ist). Die nächsten drei Argumente sind Bitfelder; man setzt das 23. Bit im 1. Array (`readfds`, read file descriptors), wenn auf auf dem File Deskriptor 23 Daten lesen möchte. Das 2. Array ist für Schreiben, das 3. für Fehlerbehandlung. Als letztes Argument übergibt man noch, wie lange man höchstens warten möchte.

Mit `select()` lassen sich offensichtlich nicht nur Timeouts implementieren, sondern man kann auch mehrere Deskriptoren auf einmal zum Lesen oder Schreiben benutzen, d.h. man kann einen Server schreiben, der mehr als eine Verbindung gleichzeitig behandeln kann.

`select()` hat aber auch wichtige Nachteile:

- `select()` sagt nicht, wie lange es denn tatsächlich gewartet hat. Man muß also noch mal `gettimeofday()` o.ä. aufrufen.

- `select()` arbeitet mit Bitfeldern. Die Größe hängt vom Betriebssystem ab. Wenn man Glück hat, kann man so 1024 Deskriptoren ansprechen, häufig noch weniger.

Unter Linux gibt `select()` doch zurück, wie lange es gewartet hat, indem es den Timeout-Wert um den Betrag senkt, den es gewartet hat. Das ist nicht nur unportabel, es beschädigt auch portable Software, die sich darauf verläßt, daß der Timeout-Wert konstant bleibt. Die Bitfeld-Größe ist unter Linux 1024, bei NetBSD 256.

Das Problem mit `select()` ist schlimmer als es aussieht, weil es manchmal von irgendwelchen Unter-Bibliotheken benutzt wird. Apache hat an dieser Stelle z.B. Probleme mit den DNS-Bibliotheken bekommen und hält sich daher von Hand die Filedeskriptoren bis 15 frei, indem neue Deskriptoren nach oben verschoben werden (das kann man mit `dup2` machen). DNS hört sonst einfach zu funktionieren auf, sobald man mal alle Deskriptoren bis 1024 belegt hat.

Eine Variation auf `select()` ist `poll()`, eingeführt 1986 mit System V Release 3. Man benutzt es ziemlich ähnlich wie `select()`:

```
struct pollfd pfd[2];
pfd[0].fd=0; pfd[0].events=POLLIN;
pfd[1].fd=1; pfd[1].events=POLLOUT|POLLERR;
if (poll(pfd,2,1000)==0) /* 2 records, 1000 milliseconds timeout */
  handle_timeout();
if (pfd[0].revents&POLLIN) can_read_on_fd_0();
if (pfd[1].revents&POLLOUT) can_write_on_fd_1();
```

`poll()` hat kein Limit auf die Anzahl oder Größe der Deskriptoren, dafür wird es auf manchen Museumsexponaten nicht unterstützt. Es gibt sogar ein aktuelles Unix-Derivat, das kein `poll()` unterstützt, und das es damit unmöglich macht, auf ihm portabel skalierbare Serversoftware laufen zu lassen: MacOS X. Von der Benutzung von MacOS X kann daher nur abgeraten werden, bis jemand dem Hersteller fundamentales Unix-Wissen vermitteln konnte.

Während man mit `poll()` überhaupt erst mal in die Lage versetzt wird, Programme mit 10.000 offenen Client-Verbindungen zu schreiben, kann ein solches Programm mit `poll()` nicht effizient laufen. Der Kernel muß das gesamte Array aus dem User-Space lesen, die zugehörigen Socket-Objekte aus seinen Datenstrukturen heraussuchen, auf neuen Daten oder Schreibbarkeit prüfen, und dann im User-Space das entsprechende Bit setzen. Heutige Prozessoren sind weitgehend von der Speicherbandbreite limitiert; das langwierige Durchgehen von großen Arrays im User-Space ist daher sehr ineffizient, insbesondere wenn sich an dem Inhalt des Arrays seit dem letzten Aufruf von `poll()` gar nichts geändert hat.

Das klingt auf den ersten Blick unvermeidlich, aber wenn man sich typische Server-Lasten mit 10.000 Verbindungen ansieht, dann sind nur wenige Verbindungen davon aktiv, die meisten sind sehr langsam oder gar ganz inaktiv. Ein viel effizienteres Schema wäre daher, wenn man die Verwaltung der Liste der gewünschten Events dem Kernel überläßt und ihm nur Veränderungen mitteilt.

Es ist normalerweise nicht so kritisch, wenn `poll()` langsam ist, weil die Events ja nicht verloren gehen, sie kommen nur später an. Der Durchsatz leidet daher nicht so stark wie die Latenz. Ich habe mit `poll()` eine bis zu um den Faktor 20 höhere Latenz als mit SIGIO oder epoll gemessen.

Das Multithreading-Debakel

In den letzten Jahren hat sich Multithreading als Lösung für alle Probleme in den Köpfen etabliert, insbesondere für Skalierbarkeit und Performance auch und vor allem bei Netzwerkprogrammierung. Die Realität sieht anders aus. Die meisten Systeme haben ein hartes Limit für die Anzahl der gleichzeitigen Threads. Eine übliche Größenordnung ist 1024. Von Skalierbarkeit kann unter solchen Umständen keine Rede sein. Auch ansonsten kann man mit Thread bizarre Effekte erleben: pthread_create kann keine Threads anlegen, selbst wenn nur bereits

4 laufen, oder manchmal sogar wenn gar keine anderen Threads laufen, solange vorher viele Threads erzeugt wurden. Es ist mir unbegreiflich, wie Leute Threads für Netzwerkprogrammierung verwenden können.

Das Multiprozeß-Modell hat den Vorteil, daß es fehlertolerant arbeitet. Wenn ein Server-Prozeß abstürzt, laufen die anderen weiter. Es gibt auch keine Memory Leaks: wenn sich ein Server-Prozeß beendet, gibt das System alle Resourcen automatisch frei. Bei Threads ist beides nicht so. Wenn ein Thread einen Absturz erzeugt, schießt das System den ganzen Prozeß mit allen anderen Threads mit ab. Multithreading-Programme haben daher häufig die Eigenschaft, schrankenlos vor sich hin zu wachsen. Manch kommerzielle Software kommt daher mit einem Shell Script, das sie nachts automatisch abschießt und neu startet, um den Speicherlecks entgegen zu arbeiten.

Man muß sich natürlich auch bei `poll`-basierten Servern darum kümmern, daß man keinen Speicher leckt, insofern ist das nicht den Threads alleine anzulasten, aber viele Multithread-Projekte sind aus Multiprozeß-Projekten hervorgegangen und kriegen ihre Lecks nie wieder in den Griff.

Asynchroner I/O

`poll` und `select` funktionieren nicht auf Dateien, nur auf Sockets, Gerätedateien und Pipes. Das ist bei Festplatten normalerweise nicht so schlimm, aber wenn man einen Server schreibt, der per NFS gemountete Dateien frei geben soll, dann blockt bei einem NFS-Timeout der gesamte Server, selbst wenn die anderen Requests alle Dateien von der lokalen Festplatte betreffen.

Die Unix-Standardisierungsgremien haben sich daher ein Modell für asynchronen I/O ausgedacht, bei dem das `read`-Äquivalent sofort zurück kehrt und man später ein Signal kriegt, wenn die Operation fertig ist (analog natürlich für `write`). Das Problem ist, daß dieses API fast niemand implementiert; Solaris hat die Funktionen, die liefern aber alle Fehler zurück (Solaris hat auch ein funktionierendes proprietäres API). Linux glibc implementiert diese Funktionen, indem pro Operation ein Thread aufgemacht wird. Weil das so fürchterlich ist, benutzt dieses API so gut wie niemand.

Asynchroner I/O hätte dabei durchaus Vorzüge gegenüber normalen Lesen. Wenn man z.B. 1000 Blöcke aus einer Datenbank lesen will, und man liest sie alle sequentiell, dann muß das System sie in der gewählten Reihenfolge lesen. Da das User-Space Programm keine Ahnung haben kann, wie die Blöcke physikalisch auf der Platte verteilt sind, muß das System unnötig die Schreib-Lese-Köpfe hin- und herbewegen. Asynchroner I/O hingegen erlaubt dem System, die Blöcke in der effizientesten Reihenfolge zu lesen. Es gibt Ansätze, asynchronen I/O auch in Linux anständig zu implementieren.

Linux 2.4: SIGIO

Linux 2.4 kann poll-Events auch über Signals mitteilen. Die Idee ist, daß man mit einem `fcntl` auf den File Deskriptor dem System mitteilt, daß man an Statusänderungen interessiert ist, und dann kriegt man immer ein Signal, wenn sich an dem Deskriptor etwas tut. Der Vorteil ist, daß das im System mit O(1) implementierbar ist. Hier ist Beispiel-Code:

```
int sigio_add_fd(int fd) {
  static const int signum=SIGRTMIN+1;
  static pid_t mypid=0;
  if (!mypid) mypid=getpid();
  fcntl(fd,F_SETOWN,mypid);
  fcntl(fd,F_SETSIG,signum);
  fcntl(fd,F_SETFL,fcntl(fd,F_GETFL)|O_NONBLOCK|O_ASYNC);
}

int sigio_rm_fd(struct sigio* s,int fd) {
  fcntl(fd,F_SETFL,fcntl(fd,F_GETFL)&(~O_ASYNC));
}
```

Signal-Handler bekommen traditionell die Signal-Nummer übergeben. Bei Linux gibt es noch ein zweites Argument, eine struct, in der u.a. der File Deskriptor drin steht, so daß man für 1000 Deskriptoren Signals über den gleichen Handler bearbeiten lassen kann.

Aus Performance-Gründen kann man auf die Zustellung der Signals auch verzichten und stattdessen das gewählte Signal blocken. Es ist dann trotzdem noch möglich, dieses Signal zu empfangen, indem man in einer Schleife `sigtimedwait` aufruft. Da kann übrigens man auch noch einen Timeout angeben, was das Timeout-Handling vereinfacht.

SIGIO ist ein sehr elegantes API, das allerdings ein Umdenken erfordert. Da man nirgendwo sagt, an welchen Events man interessiert ist, kriegt man alle Events zugestellt. Bei `select` und `poll` ist es so, daß man auf ein Event auch einfach nicht reagieren kann. Der nächste `poll`-Aufruf signalisiert dann das selbe Event noch einmal. So kann man sehr schön für Fairness sorgen und Rate Limiting implementieren. SIGIO hingegen teilt nur das erste Event mit. Wenn man darauf nicht reagiert, wird man nie wieder auf diesen Deskriptor hingewiesen werden.

Die Angelegenheit ist sogar noch etwas schwieriger: Wenn man bei `poll` ein Lese-Event bekommt, `read` aufruft, 100 Bytes liest, und wieder `poll` aufruft, funktioniert das. Bei SIGIO muß man `read` so lange aufrufen, bis man explizit `EAGAIN` als Fehlermeldung kriegt ("später nochmal probieren"). Diese Fehlermeldung kann `read` nur erzeugen, wenn man non-blocking I/O für diesen File Deskriptor aktiviert hat:

```
#include <fcntl.h>

fcntl(fd,F_SETFL,fcntl(fd,F_GETFL,0) | O_NDELAY);
```

Das `poll`-Modell nennt man level triggered, das SIGIO-Modell heißt edge triggered. In der Praxis ist ersteres für Programmierer einfacher zu verstehen. Das SIGIO-Modell hat aber den Vorteil, daß sich das Programm länger am Stück mit der gleichen Verbindung beschäftigt, so daß die zugehörigen Daten alle im Cache sind. So ist es geringfügig effizienter. Man kann auch mit `poll` arbeiten, als wäre es edge triggered, aber nicht umgekehrt.

```
for (;;) {
  timeout.tv_sec=0;
  timeout.tv_nsec=10000;
  switch (r=sigtimedwait(&s.ss,&info,&timeout)) {
  case -1: if (errno!=EAGAIN) error("sigtimedwait");
  case SIGIO: puts("SIGIO!  overflow!"); return 1;
  }
  if (r==signum) handle_io(info.si_fd,info.si_band);
}
```

Aber auch SIGIO hat Nachteile: der Kernel reserviert pro Prozeß einen Speicherbereich für die Liste der ausstehenden Events. Wenn dieser Speicherbereich voll ist, würden Events verloren gehen. Der Kernel signalisiert daher dann mit SIGIO (mit dem Signal namens SIGIO, nicht dem Konzept), daß die Queue vollgelaufen ist. Das Programm muß dann die Queue manuell leeren, indem für den Signal Handler kurzzeitig `SIG_DFL` eingetragen wird, und sich die Events mit `poll` abholen.

In der Praxis heißt das, daß man bei SIGIO eben doch noch ein Array mit `struct pollfd` verwalten muß; das möchte man sich ja aber eigentlich gerade sparen, deshalb benutzt man ja SIGIO! Das klingt vielleicht auf den ersten Blick nicht nach Arbeit, aber pollfds müssen kontinuierlich hintereinander liegen; man kann nicht in der Mitte des Arrays einen leeren Eintrag haben, sonst bricht `poll` mit `EBADFD` ab. Wenn mein Server also viele Verbindungen offen hat, und in der Mitte wird eine geschlossen, muß ich mein Array anpassen. Das heißt aber auch, daß ich den zugehörigen Eintrag im Array nicht einfach finden kann, indem ich den Deskriptor als Index nehme! Daraus folgt, daß man O(1)-Finden des zugehörigen Eintrags nur über eine Indirektion mit einem zweiten Array implementieren kann, und der Code wird dann entsprechend unübersichtlich.

Es gibt noch einen Nachteil von SIGIO: man hat einen Syscall pro mitgeteiltem Event. Bei poll ist zwar die Abarbeitung des Syscalls teuer, aber dafür hat man da nicht viele von. Unter Linux ist der Aufruf eines Syscalls sehr effizient gelöst, daher muß man sich da nicht viel Sorgen machen; die kommerziellen Unixen haben da teilweise mehr als eine Größenordnung größere Latenzen. Aber unabhängig davon, ob das jetzt sehr oder nur mäßig viel verschwendete Leistung ist, man kann sich auch ein API vorstellen, bei dem man mehr als ein Event auf einmal gemeldet bekommen kann.

Solaris: /dev/poll

Bei Solaris kann man seit zwei-drei Jahren poll beschleunigen, indem man das Device `/dev/poll` öffnet, dort sein `pollfd`-Array hinein schreibt, und dann mit einem `ioctl` auf Events wartet.

Der `ioctl` sagt einem, wie viele Events anliegen, und so viele `struct pollfd` liest man dann von dem Device. Man hat also nur für die Deskriptoren Aufwand, bei denen auch tatsächlich Events anliegen. Das spart sowohl im Kernel als auch im User-Space gewaltig Aufwand und ist auch mit minimalem Aufwand auf bestehende Programme portierbar.

Es gab Ansätze, ein solches Device auch unter Linux zu implementieren. Keiner von ihnen hat es in den Standard-Kernel geschafft, und die Patches verhielten sich unter hoher Last teilweise undeterministisch, daher ist dieses API auf Solaris beschränkt.

Linux: epoll

Für Linux 2.4 gibt es einen Patch, der `/dev/epoll` implementiert. Die Anwendung entspricht weitgehend der Solaris-Variante:

```
int epollfd=open("/dev/misc/eventpoll",O_RDWR);
char* map;
ioctl(epollfd,EP_ALLOC,maxfds); /* Hint: Anzahl der Deskriptoren */
map=mmap(0, EP_MAP_SIZE(maxfds), PROT_READ, MAP_PRIVATE, epollfd, 0);
```

Man fügt ein Event an, indem man auf das Device einen `pollfd` schreibt. Man löscht ein Event, indem man auf das Device einen `pollfd` mit `events==0` schreibt. Wie bei Solaris holt man die Events mit einem `ioctl` ab:

```
struct evpoll evp;
for (;;) {
  int n;
  evp.ep_timeout=1000;
  evp.ep_resoff=0;
  n=ioctl(e.fd,EP_POLL,&evp);
  pfds=(struct pollfd*)(e.map+evp.ep_resoff);
  /* jetzt hat man n pollfds mit Events in pfds */
}
```

Dank `mmap` findet hier überhaupt kein Kopieren zwischen Kernel und User-Space statt. Dieses API hat es nie in den Standard-Kernel geschafft, weil Linus `ioctl` nicht mag. Seine Begründung ist, daß das ein Dispatcher sei, und man habe über die Syscall-Nummer ja schon einen Dispatcher, und daher solle man doch bitte den benutzen. Der Autor von epoll hat die Patches daraufhin noch einmal mit Syscalls implementiert, und diese API ist seit 2.5.51 in der dokumentierten Form im Standard-Kernel enthalten.

```
int epollfd=epoll_create(maxfds);
struct epoll_event x;
x.events=EPOLLIN|EPOLLERR;
x.data.ptr=whatever; /* hier trägt man sich einen Cookie ein */
epoll_ctl(epollfd,EPOLL_CTL_ADD,fd,&x);
/* ändern ist analog */
epoll_ctl(epollfd,EPOLL_CTL_MOD,fd,&x);
/* löschen ist analog, nur fd muß eingetragen sein */
epoll_ctl(epollfd,EPOLL_CTL_DEL,fd,&x);
```

Die EPOLLIN, ... Konstanten sind im Moment identisch mit POLLIN, aber der Autor wollte sich da alle Optionen offen halten. Das epoll-API sagt einem nicht automatisch den Deskriptor zu dem Event. Wenn man das möchte, muß man den Deskriptor als Cookie eintragen. Der Cookie ist ein 64-bit Wert, hier kann man also auch einen

Zeiger auf eine Datenstruktur eintragen, in der dann neben anderen Daten auch der Deskriptor steht. Seine Events holt man sich dann mit `epoll_wait` ab:

```
for (;;) {
  struct epoll_event x[100];
  int n=epoll_wait(epollfd,x,100,1000); /* 1000 Millisekunden */
  /* x[0] .. x[n-1] sind die Events */
}
```

Windows: Completion Ports

Auch Microsoft muß skalierbare Netzwerk-Server anbieten können. Die Microsoft-Lösung hierfür ist, daß man einen Thread Pool aufmachen, und dann in jedem Thread mit einem SIGIO-ähnlichen Verfahren die Events benachrichtigt bekommt.

Dieses Verfahren verbindet die Nachteile von Threads mit den Nachteilen von SIGIO. Leider gibt es unter Windows keine Alternativen. Für die Neugierigen: `select()` wird unter Windows über ein leeres Off-Screen Fenster implementiert, das dann über den Fenster Input-Event Mechanismus die Netzwerk-Events mitgeteilt bekommt.

FreeBSD: kqueue

kqueue ist eine Kreuzung aus epoll und `/dev/poll`. Man kann sich aussuchen, ob man nach level oder egde getriggert werden will. Außerdem hat man noch eine Art SIGIO und File und Directory Status Notification eingebaut. Dabei ist das API leider ziemlich unübersichtlich geworden. Von der Performance her ist kqueue mit epoll vergleichbar.

Sonstige Tricks zur Performance-Steigerung

writev, TCP_CORK und TCP_NOPUSH

Ein HTTP-Server schreibt erst einen (kleinen) Antwort-Header und dann die Datei in den Socket. Wenn er für beides einfach `write()` aufruft, erzeugt das ein kleines TCP-Paket für den Header, und danach noch ein TCP-Paket für die Daten. Wenn man nur eine kleine Datei per HTTP übertragen will, hätte beides vielleicht in ein Paket gepaßt.

Offensichtlich kann man einen Puffer nehmen, und in diesen erst den Header und dann den Dateiinhalt hineinkopieren, und dann den Puffer schreiben. Das ist aber Verschwendung von RAM-Bandbreite. Es gibt drei andere Lösungen: `writev()`, `TCP_CORK` (Linux) und `TCP_NOPUSH` (BSD).

`writev()` ist wie ein Batch-`write()`. Man übergibt ein Array mit Puffern, `writev()` schreibt sie alle hintereinander. Der Unterschied ist normalerweise nicht meßbar, aber bei TCP-Servern macht er sich bemerktbar.

```
struct iovec x[2];
x[0].iov_base=header; x[0].iov_len=strlen(header);
x[1].iov_base=mapped_file; x[1].iov_len=file_size;
writev(sock,x,2); /* returns bytes written */
```

`TCP_CORK` ist eine Socket-Option, die man mit `setsockopt` vor dem Schreiben setzt und nach dem letzten Schreiben wieder zurück setzt. Das sagt dem Kernel metaphorisch, daß man die TCP-Flasche verkorkt hält, und erst am Ende löst man den Korken und alles sprudelt auf einmal über das Netz hinaus.

```
int null=0, eins=1;
```

```
/* Korken reinstecken */
setsockopt(sock,IPPROTO_TCP,TCP_CORK,&eins,sizeof(eins));
write(sock,header,strlen(header));
write(sock,mapped_file,file_size);
/* Korken ziehen */
setsockopt(sock,IPPROTO_TCP,TCP_CORK,&null,sizeof(null));
```

`TCP_NOPUSH` ist das BSD-Äquivalent zu `TCP_CORK`. Der einzige Unterschied ist, daß man `TCP_NOPUSH` vor dem letzten Schreiben zurück setzen muß, nicht danach.

sendfile

Typische hochskalierbare Server lesen immer Daten aus Dateien und schreiben sie über das Netzwerk hinaus. Dabei hält der Server-Prozeß einen Puffer vor, in den er die Daten aus der Datei liest, und den er dann über das Netz schreibt. Hier werden also Daten unnötig kopiert. Es wäre effizienter, wenn man die Datei direkt versenden könnte.

Das haben sich auch diverse Unix-Hersteller gedacht, und Microsoft hat einen solchen Syscall auch in Windows eingebaut. Die meisten Server-Netzwerkkarten (alle Gigabit-Karten) beherrschen Scatter-Gather I/O, sozusagen `writev` in Hardware. Das Betriebssystem kann dann die IP, TCP und Ethernet-Header in einem Puffer zusammenbauen, und die Netzwerkkarte holt sich den Inhalt des Pakets direkt aus dem Buffer Cache. Dieses Zusammenspiel nennt man zero copy TCP und es ist sozusagen der heilige Gral der Netzwerkprogrammierung.

Man kann die Daten aus dem Buffer Cache auch per `mmap` direkt in den User-Space Speicher einblenden, ohne daß Daten kopiert werden müßten. Leider kann Linux bei einem `write()` auf einen solchen Speicherbereich kein zero copy TCP initiieren, dazu muß man `sendfile` benutzen. Trotzdem macht es Sinn, alle Daten wenn möglich mit `mmap` statt `read` zu lesen, weil man so im System Speicherplatz spart, den das System dann zum Cachen verwenden kann.

Das große Problem bei `sendfile()` ist, daß es jeder Anbieter mit einer anderen Semantik und anderen Argumenten implementiert hat. Nicht einmal die freien Unixe haben sich an das gleiche Format gehalten.

Sonstige Performance-Tweaks

Auf heutiger Hardware ist `fork()` sehr effizient implementiert. Die Datenbereiche der Prozesse werden nicht mehr kopiert, sondern es werden nur noch die Page-Mappings übernommen. Das ist Größenordnungen effizienter als früher, aber bei einem Server-Prozeß mit 20000 gemappten Dateien kann es sich schon um einen meßbaren Zeitraum handeln. Es macht daher auch heute noch Sinn, in seinen Servern (z.B. für den Aufruf von CGIs) `vfork()` vorzuziehen.

Bei Servern mit sehr vielen offenen Deskriptoren kann `open()` plötzlich meßbar langsamer werden. Das liegt daran, daß POSIX vorschreibt, daß immer der kleinste unbelegte Deskriptor genommen werden muß. Das implementiert der Kernel, indem er Bitfelder linear durchgeht. Es ist also theoretisch denkbar, daß man ein paar Nanosekunden sparen kann, wenn man die untersten Dateideskriptoren mit `dup2` frei hält. Diese Idee geistert seit Jahren über diverse Mailinglisten; mir ist aber kein Fall bekannt, wo das tatsächlich mal experimentell nachgewiesen worden ist.

Auf neueren x86-Prozessoren gibt es neben dem Standard-Syscall-Mechanismus über Software Interrupt 80 auch noch einen anderen Syscall-Mechanismus. Der Linux Kernel hat relativ frischen Support dafür, indem er neben dem Programm im User Space noch eine weitere Code Page mit dem neuen Syscall einblendet. Die Adresse der Page wird Programmen über den ELF AUX Vektor übergeben (Daten auf dem Stack hinter `argv` und dem Environment). Meinen Messungen zufolge reduziert sich damit die Syscall-Latenz auf die Hälfte. In den HTTP-Benchmarks war die Auswirkung aber mit 2-5% nur knapp über der statistischen Rauschgrenze.