Skalierbare Netzwerkprogrammierung

Felix von Leitner
felix-ccc@fefe.de

27. Dezember 2002

Zusammenfassung

Wie schreibt man Netzwerk-Server, die auch mit 10000 Klienten klar kommen? Welche Flaschenhälse gibt es und wie umgeht man sie?

Achtung: es geht hauptsächlich um Linux!

Netzwerk-Programmierung Basics

Unix: Geräte und IPC-Mechanismen werden wie Dateien angesprochen. Dateien werden über Integer angesprochen.

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

Das API ist trivial und offensichtlich. `read()` kehrt zurück, wenn die Daten gelesen wurden. `write()` kehrt zurück, wenn die Daten (in den Buffer Cache des Betriebssystems) geschrieben wurden.

Netzwerk-Programmierung Basics

Sockets funktionieren analog (hier ein HTTP-Client):

```
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);
```

Netzwerk-Programmierung Basics

Ein Server funktioniert ähnlich:

```
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);
```

Server mit mehr als einem Client?

Wie kann ein Server mehr als einen Client verwalten?

Sobald er von einem Client Eingaben liest, bleibt der Prozeß stehen, bis auch tatsächlich Eingaben vorliegen!

Traditionelle Lösung: pro Client einen neuen Prozeß aufmachen.

Probleme mit einem Prozeß pro Client

1. Scheduler sind für den Fall mit wenigen ($<$100) Prozessen optimiert. Dieses Modell zwingt das System bei jedem blockenden `read()` in den Scheduler.

2. Prozesse belegen normalerweise signifikante Mengen an Speicherplatz. 10000 leere Verbindungen bringen auch größere Server zum Swappen.

3. Bei manchen Systemen (Solaris, Windows) ist die Prozeßerzeugung haarsträubend ineffizient.

4. Wie realisiert man Timeouts?

Scheduler

Der Scheduler ist der Teil des Betriebssystems, der entscheidet, welcher Prozeß als nächstes laufen soll. Typischerweise idlen eine handvoll Prozesse im System herum, während ein bis zwei Prozesse laufen wollen.

Linux unterbricht laufende Prozesse jede 1/100 Sekunde (Alpha: 1/1000), um anderen Prozessen eine Chance zu geben. Der Scheduler wird aufgerufen, wenn ein Prozeß blockt (weil er auf Eingaben wartet), sein CPU-Zeitquantum verbraucht hat oder sich beendet.

Der Scheduler muß dann von den lauffähigen Prozessen den aussuchen, der es am meisten verdient hat. Es macht also Sinn, die Prozesse in zwei Listen zu halten: eine für die, die laufen wollen, und eine für den Rest.

Scheduler

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.

Bei 10000 Prozessen trasht diese Berechnung den 2nd Level Cache.

Außerdem sind diese Kernel-Locks Spinlocks, d.h. die zweite CPU begibt sich in eine Schleife, bis der Lock freigegeben wurde. Die typische Lösung bei den Kommerz-Unixen ist, daß man eine Run Queue pro CPU hat. Desweiteren kann man die Run Queue noch sortiert halten, z.B. mit einem Heap als Datenstruktur, oder man kann pro Priorität eine Run Queue haben.

Scheduler: Linux 2.4

Der Linux Scheduler hat eine einzelne unsortierte Run Queue mit allen laufen wollenden Prozessen und eine Task-Liste mit allen schlafenden Prozessen und Zombies. Alle Operationen auf der Run Queue sind hinter einem Spinlock. Der Scheduler bemüht sich um CPU-Affinität.

Es gibt für Linux diverse neue Scheduler, mit Heaps und multiplen Queues, aber der erstaunlichste Scheduler ist aber der O(1) Scheduler von Ingo Molnar.

Der O(1) Scheduler ist in den 2.5 Hacker-Kerneln Default.

Der O(1) Scheduler

Der O(1) Scheduler hat pro CPU zwei Arrays mit einer verketteten Liste pro Priorität. Um die nichtleere Queue mit der höchsten Priorität zu finden, gibt es noch eine Bitmap (auf x86 gibt es eine bit-search Instruktion). Das eine Array hat die laufenden Prozesse. Fertig gelaufene Prozesse werden in das andere Array eingetragen. Wenn das Array alle ist, werden die Zeiger auf die Arrays getauscht.

Prozesse werden bestraft, wenn sie mehr CPU haben wollen, als ihnen zusteht. Das ist im Gegensatz zum Belohnen von interaktiven Prozessen eine O(1) Operation.

Dieser geniale Scheduler skaliert praktisch unabhängig von der Anzahl der Prozesse. In Tests haben Leute 100.000 Threads aufgemacht und keine Verlangsamung beobachtet.

Kontextwechsel

Bei Prozessoren mit logischer Cache-Adressierung muß man beim Kontextwechsel die Caches flushen. Das betrifft vor allem ARM. Auf manchen Artitekturen (ARM again) muß man auch noch manuell die Pipeline leeren.

Alle Architekturen müssen beim Kontextwechsel den TLB flushen und die Register sichern und laden. Da der x86 nur so wenige Register hat und den Cache nicht flushen muß, ist hier der Kontextwechsel besonders effizient.

Was hat das mit Netzwerkprogrammierung zu tun?

Hier ist ein vmstat eines MPEG Audio players:

```
   procs                  memory      swap          io     system      cpu
 r  b  w   free buff  cache   si   so    bi    bo   in    cs us sy id
 0  0  0  60316 9136 316280    0    0   135   113   47     8  8  9 82
 0  0  0  60296 9152 316280    0    0     0     0  200    54  3  0 97
```

Jetzt mit einem http-Bench über loopback und eth0:

```
 4  0  0  56136 9236 317936    0    0    24     0  228  7953  6 59 35
 0  0  0  56308 9236 317936    0    0     0     0  221  1590  4 10 86

 4  0  0  53476 9304 320180    0    0     0     0 16416 12319 26 74  0
 5  0  0  52552 9304 320180    0    0     0     0 16391 12307 20 80  0
```

Was hat das mit Netzwerkprogrammierung zu tun?

Die Anzahl der Kontextwechsel geht bei einem Netzwerk-Server unter Last dramatisch nach oben. Hier zählen nicht die Übergänge zwischen User Space und System, nur die Übergänge zwischen zwei Prozessen!

D.h. bei jedem Kontextwechsel entscheidet der Scheduler, welcher Prozeß jetzt laufen soll. Bei 12000 Scheduler-Aufrufen pro Sekunde kann man sich vorstellen, daß hier auch kleine Effizienzunterschiede entscheiden.

Speicherplatz und Prozeßerzeugungskosten

Der Speicherplatz für einen leeren Prozeß ist bei aktuellen Linux-System in der Tat erschreckend groß. Schuld ist aber nicht Linux, sondern die libc unter Linux, die GNU libc.

Diese führt mit großem Abstand alle libc-Implementationen in Sachen Bloat und Speicher-Verschwendung an. Die Schmerzen sind eines Tages so groß geworden, daß ich eine neue libc geschrieben habe. Mit dieser kann man auf einem Athlon Hallo Welt in einem statischem Binary von 300 Bytes haben, inklusive fork, exec und wait kostet die Prozeßausführung ca. 200000 CPU-Zyklen.

Ein 2 GHz Atlon könnte also rein rechnerisch pro Sekunde 10000 Prozesse erzeugen. Das reicht für typische Skalierbarkeitsanforderungen.

Speicherplatz und Prozeßerzeugungskosten

Im Solaris-Lager sieht es weniger rosig aus. Die Prozeßerzeugungskosten sind geradezu legendär groß. Teilweise ist das auf Hardware-Eigenschaften zurück zu führen, weil SPARC nicht 32 sondern 1024 Register hat („Register Windows“).

Das erklärt aber nicht, wieso Linux auf der selben Hardware schneller Prozesse erzeugen kann als Solaris Threads. Solaris ist einfach auffallend schlecht und für skalierbare Netzwerk mit den Mehrprozeßmodell (und dem Multithread-Modell) nicht geeignet.

Wie implementiert man Timeouts?

Timeouts kann man am einfachsten über `alarm()` implementieren. Alarm schickt dem Prozeß nach n Sekunden ein Signal. Wenn man das nicht abfängt, killt es den Prozeß. Das ist geradezu ideal für triviale Netzwerk-Server.

Timeouts mit select()

select wartet auf Ereignisse an einem oder mehreren Filedeskriptoren.

```
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();
```

Der Timeout kann zwar mit Mikrosekunden-Genauigkeit angegeben werden, aber so hoch auflösend implementiert das kein Unix. Das erste Argument ist der höchste Filedeskriptor plus eins. (Yuck!)

Nachteile von select()

select sagt einem nicht, wie lange es auf Events gewartet hat. Man muß also nochmal von Hand `gettimeofday()` aufrufen, um grobkörnigere Timeouts (pro Kommando, pro Header, pro Megabyte) zu implementieren.

select arbeitet mit Bitfeldern. Die Größe ist je nach Betriebssystem anders. Wenn man Glück hat, kann man so 1024 Deskriptoren ansprechen, häufig noch weniger.

Das ist ärgerlicher als es aussieht, weil manche schlechten DNS-Libraries select für ihr Timeout-Handling benutzen. Wenn man vorher 1024 Dateien aufgemacht hat, geht plötzlich DNS nicht mehr, weil der File Deskriptor über 1024 ist! Apache hält daher die Filedeskriptoren bis 15 frei. (Igittigitt!)

Timeouts mit poll()

poll ist eine einfache select-Variation:

```
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();
```

Vorteil: Kein Limit auf Anzahl der Records und Filedeskriptoren.

Nachteil: poll gibt es auf einigen obsoleten Museums-Exponaten nicht.

Nachteile von poll()

Das gesamte Array muß ständig zwischen User- und Kernel-Space hin- und herkopiert werden. Der Kernel findet dann jeweils heraus, ob ein Event anliegt und setzt revents. Heutige Prozessoren verbringen eh die meiste Zeit damit, auf den Speicher zu warten. poll verschärft das noch, und die Last steigt linear mit der Anzahl der Deskriptoren.

Das ist nicht ganz so schlimm, wie es aussieht, denn wenn poll länger braucht, gehen ja keine Events verloren, sondern es sind nur das nächste Mal mehr revents gesetzt. Der Server ist also nicht plötzlich nicht in der Lage, die Last zu tragen, aber die Latenz zwischen Anfrage und Reaktion nimmt zu.

Threads

Wie erwähnt haben einige Systeme Performance-Probleme bei ihrer Prozeßerzeugung.

Wenn der Hersteller dann mehr Fachleute im Marketing als in der Entwicklung sitzen hat, wird so ein Problem schon mal so gelöst, daß man alle Programme umschreiben muß, die Programme danach viel schwieriger zu debuggen sind (so kann man auch noch neue Debugger und Schulungen verkaufen) und letztlich jeder nur Nachteile hat.

So ist es unter Solaris und Windows NT geschehen, als die Hersteller Multithreading erfunden haben. Als Sun niemanden überzeugen konnte, Kompatibilität für ein neues API mit fragwürdigem Vorteil zu opfern, haben sie ihr API halt in ein „Standardgremium“ eingebracht und mit dem Java-Hype neue Maßstäbe gesetzt.

Threads

Was hat Sun von Threads? Multithreading performt besser, wenn man mehr CPUs kauft. Sun verkauft CPUs.

Das Problem bei Threads ist, daß auf schlechten Systemen natürlich auch das Erzeugen von Threads langsam ist. Solaris und Windows können zwar schneller Threads erzeugen als Prozesse, aber performant ist das immer noch nicht.

Also hat man Thread Pools eingeführt. Die Idee ist, daß man am Anfang n Threads aufmacht (sagen wir, 128), und die teilen sich die Verbindungen. Wenn mehr als 128 Verbindungen reinkommen, müssen die halt warten. Dafür entfallen die Kosten für die Thread-Erzeugung. Ich verstehe unter Skalierbarkeit allerdings was anderes.

Threads

Threads haben auch Vorteile: durch ihre Ineffizienz erzeugen sie massive Skalierbarkeitsprobleme. Das führt dazu, daß wir neue, schnellere Hardware brauchen. Auch die RAM-Preise sind auf einem Rekord-Tief.

Und auch auf der Softwareseite wird die Innovation auf Systemebene von den Idioten auf Anwendungsebene getrieben. Weil Lotus Notes so unglaublich ineffizient ist und pro Client eine Verbindung offen hält, hat IBM bei Linux den Code für den Fall „ein Prozeß, viele offene Verbindungen“ contributed. Und der O(1)-Scheduler ist ursprünglich für irgendeinen irrelevanten Java Benchmark geschrieben worden.

Letztlich kommt also dieser ganze Bloat uns allen zu Gute. Wir müssen nur dafür sorgen, daß es auch Alternativen zu dem üblichen Applikations-Bloat gibt.

Threads

Das Multiprozeß-Modell hat den Vorteil, daß Fehler verziehen werden. Es gibt keine Memory Leaks; bei Prozeßende gibt das System alle Resourcen automatisch frei.

Wenn ein Prozeß segfaulted, laufen die anderen weiter.

Bei Threads ist der Programmierer gezwungen, Memory Leaks zu vermeiden. Weil die meisten Idioten-Programmierer dafür zu dämlich sind, gibt es bei Java (und Visual Basic) Garbage Collection. Wenn dann ein Java-Depp mal in C oder C++ einen Server schreiben will, erlebt er sein Waterloo.

Ich persönlich sehe das als Vorteil von Threads. Außer durch externe Schmerzeinwirkung lernen die Leute nicht.

Asynchroner I/O

POSIX spezifiziert ein API für asynchronen I/O. Leider ist das API nur in der Minderzahl der Systeme implementiert, bei Linux z.B. nicht. Die glibc hat eine Emulation, die pro Request einen Thread aufmacht. Das ist noch viel schlechter als alle anderen Varianten, daher benutzt dieses API kein Mensch.

Die Idee ist, daß man (fd,offset,buf,sizeof buf) anmelden kann, und dann kann man später nachfragen, ob er schon fertig ist, oder sich ein Signal geben lassen, wenn er fertig ist.

Nachteil: dem Signal sieht man nicht an, welcher der ausstehenden Requests gerade fertig wurde :-(

Asynchroner I/O

Im Gegensatz zu poll und select funktioniert async I/O auch mit Dateien auf der lokalen Platte. Wenn man also 1000 Blocks aus einer Terabyte-Datenbank lesen will, zwingt man dem System mit lseek() und read() die Reihenfolge und damit die Bewegungen des Plattenkopfes auf.

Bei async I/O kann das System zuerst den Block lesen, der dem Kopf im Moment am nächsten ist. Vom Prinzip her ist das prima, in der Praxis taugt es leider nichts.

Solaris hat z.B. ein eigenes proprietäres async I/O API, und sie bieten das POSIX API an, aber letzteres sind nur Stubs die ENOSYS liefern :-(

Das wichtigste System, das async I/O anbietet, ist m.W. FreeBSD (und die anderen BSDs haben es wohl übernommen oder arbeiten daran).

Linux 2.4: SIGIO

Linux 2.4 kann auch über Signals die poll-Events mitteilen.

```
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));
}
```

Linux 2.4: SIGIO

SIGIO ist *kein* direkter poll-Ersatz. poll meldet, wenn ein Deskriptor lesbar ist. SIGIO meldet, wenn sich der Lesbar-Status geändert hat.

Wenn poll POLLIN meldet, und die Anwendung liest nichts, meldet der nächste poll wieder POLLIN für den selben Deskriptor; SIGIO nicht. In der Fach-Literatur nennt man poll level-triggered und SIGIO edge-triggered.

Die Signals holt man am effizientesten mit `sigtimedwait` ab, nachdem man SIGIO und `signum` geblockt hat. Dann läuft das Abarbeiten synchron und man braucht kein Locking.

Linux 2.4: SIGIO

```
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);
}
```

`info.si_band` ist identisch zu `pollfd.revents`.

Linux 2.4: SIGIO

Aber auch bei SIGIO gibt es Nachteile. Der Kernel hat eine Queue von Events, die der Reihe nach per Signal übergeben werden.

Die Queue hat eine feste Länge, und man kann sie auch nicht einstellen. Wenn die Queue voll läuft, kriegt man ein SIGIO und keine einzelnen Events.

Dann muß man die Queue flushen, indem man für `SIG_DFL` einträgt und die aufgelaufenen Events mit poll abgreifen.

Dieses API reduziert zwar den Speicher-Traffic, aber jetzt hat man einen Syscall pro Event. Das ist auch nicht gerade der Gipfel der Effizienz. Abgesehen davon macht einem der Queue-Überlauf das Leben zur Hölle. Wenn ich SIGIO einsetze, will ich mir schon dieses eklige Gefummel mit den pollfds sparen.

Gefummel mit den pollfds?!

Die pollfds sind ein Array von Filedeskriptoren. Nehmen wir an, ich schreibe einen Webserver. In meiner Lese-Routine kriege ich signalisiert, daß die Gegenseite aufgelegt hat. Dann muß ich `close()` aufrufen und den Record aus den pollfds austragen.

Ich kann nicht einfach die Events in dem Record auf 0 setzen, weil poll sonst mit EBADFD abbricht. Also muß ich den Record mit dem letzten im Array überschreiben und die Länge des Arrays reduzieren.

Ich brauche in jedem Fall einen Index, damit ich zu einem Deskriptor den zugehörigen Record in den pollfds finde. Diesen Index muß ich dann auch updaten. Das ist widerlich fummelig. Ein paar hundert Zeilen C-Code kommen da zusammen.

/dev/poll

Bei Solaris gibt es seit zwei-drei Jahren ein neues poll-API. Man öffnet das neue Device `/dev/poll`, schreibt die pollfds mit write rein, und wartet dann mit einem ioctl auf Events.

Der ioctl sagt einem dann, wie viele Events da sind, und so viele pollfds liest man dann von dem Device. Man kriegt also nur für die Deskriptoren Meldung, bei denen auch Events vorliegen.

Das spart auch das Durchsuchen der pollfds nach denen, bei denen tatsächlich Events waren.

Es gab auch mal Ansätze, ein solches Device für Linux zu implementieren. Keiner von ihnen hat es in den Standard-Kernel geschafft, und die Patches verhielten sich unter Last auch nicht vertrauenseinflößend.

/dev/epoll

Für Linux 2.4 gibt es einen Patch, der `/dev/epoll` implementiert.

```
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.

/dev/epoll

Man holt 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 Hin- und Herkopieren zwischen Kernel- und User-Space statt.

/dev/epoll

Der Nachteil von /dev/epoll ist, daß es nur ein Patch ist. Linus mag keine neuen Devices im Kernel. Er findet, daß schon der Syscall-Dispatcher ein Dispatcher ist, man müsse da nicht noch zusätzlich mit ioctl darüber hinaus dispatchen.

Daher hat der Autor von /dev/epoll das API noch einmal mit syscalls implementiert, und dieses API ist dann auch im 2.5er Kernel gelandet (seit 2.5.51 sogar in der dokumentierten Form ;-)).

epoll

```
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.

epoll

Seine Events holt man sich dann so 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 */
}
```

In `epoll_event` steht nicht der eigentliche Deskriptor! Die Begründung war, daß man die Deskriptoren ja auch später noch mit dup2 umbenennen oder gar an andere Prozesse übertragen kann, wo sie dann eine noch andere Nummer haben.

Also nutzt man den Cookie, um sich einen Zeiger auf seine Kontext-Daten zurückliefern zu lassen, oder man trägt dort direkt den Deskriptor ein.

Windoze: Completion Ports

Auch Microsoft hat das Problem, daß sie skalierbare Netzwerkprogrammierung anbieten wollen. Die Microsoft-Lösung hierbei ist, daß man einen Thread Pool aufmacht, und dann in jedem Thread mit einem SIGIO-ähnlichen Verfahren die Events benachrichtigt bekommt.

Hier werden also die Nachteile von Threads mit den Nachteilen von SIGIO verbunden. Es ist beeindruckend, was Microsofts Marketing-Maschine aus diesem Schrumpelküken für ein Monster aufgeblasen hat.

Immerhin gibt es da auch lustige Zitate in der Dokumentation: „*First of all, threads are system resources that are neither unlimited nor cheap*.“ Äh, nicht cheap? Ich dachte, das sei ihre Existenzberechtigung!

writev, TCP_CORK und TCP_NOPUSH

Ein HTTP-Server schreibt erst einen Antwort-Header und dann die Datei in den Socket. Wenn er für beides einfach `write()` benutzt, erzeugt das ein kleines TCP-Paket für den Header, danach die Daten.

Wenn man eine sehr kleine Datei per HTTP holen will, hätte beides vielleicht in ein Paket gepaßt.

Offensichtlich kann man das Lösen, indem man die Daten puffert. Das heißt aber, daß man die Daten unnötig hin- und herkopiert. Es gibt drei andere Lösungen: `writev`, `TCP_CORK` (Linux) und `TCP_NOPUSH` (BSD).

writev

writev ist wie ein Batch-write. Man übergibt ein Array mit Puffern, writev schreibt sie einfach alle.

Der Unterschied ist normalerweise nicht meßbar, außer halt bei TCP-Verbindungen.

```
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

```
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 von BSD ist weitgehend äquivalent, nur daß man es *vor* dem letzten write wieder auf Null setzen muß, nicht danach.

FreeBSD: kqueue

kqueue ist eine Kreuzung aus epoll und `/dev/poll`. Man kann sich aussuchen, ob man seine Antworten edge oder level triggered haben möchte. Außerdem hat man auch noch SIGIO und file status notification reingehackt. Das Problem ist, daß das API dabei ziemlich unübersichtlich geworden ist.

Von den Performancedaten her sollte kqueue aber sigio ähnlich sein.

sendfile

`sendfile()` ist wie `write()` direkt von einem File Deskriptor zu einem Socket, d.h. der Puffer und das `read()` entfallen.

Der Vorteil ist, daß hier keine Daten unnötig kopiert werden. Der Kernel kann die Daten direkt aus dem Buffer Cache zur Netzwerkkarte kopieren.

Manche aktuelle Netzwerkkarten (alle Gigabit-Ethernet-Karten) beherrschen Scatter-Gather I/O, sozusagen `writev` auf Treiber-Niveau. Der Kernel kann dann also die IP und TCP Header in einem Buffer zusammenbauen und die zugehörigen Daten kopiert sich die Netzwerkkarte dann direkt aus dem Buffer Cache. Das Ergebnis heißt *zero copy TCP* und ist der heilige Gral der Netzwerk-Performance.

mmap

mmap hat zwar so direkt nichts mit Netzwerk zu tun, aber man muß es kennen, wenn man anständige Performance haben will. Man kann eine Datei unter Unix mit mmap in den Speicher laden, so daß kein Platz für Puffer verschwendet wird. Man kriegt dabei direkt die Datei aus dem Buffer Cache des Systems in seinen Adressbereich gemappt.

Bei einem Server mit 10000 Prozessen machen sich die gesparten Buffer schon bemerkbar. Typische Buffer-Größen sind 4 oder 8k. mmap spart also schon mal 40 bzw 80 MB Speicher, den das System dann zum Cachen benutzen kann.

vfork

fork ist auf heutigen Unixen sehr performant implementiert, weil da kein tatsächlicher Speicherinhalt mehr kopiert wird, sondern nur noch die Page Tables.

Trotzdem kann das eine umfangreiche Aktion werden, wenn man z.B. einen großen Prozess hat, der 2000 Dateien mit mmap in den Speicher geladen hat. In solchen Fällen macht es sich bezahlt, wenn man statt `fork` den Legacy-`vfork` benutzt.

`vfork` ist eigentlich obsolet, seit wir MMUs haben, daher ist es bei Linux auch ursprünglich nicht implementiert gewesen (bzw. als Alias auf `fork`). Seit 2.2 gibt es aber wieder einen `vfork` Syscall. Es macht Sinn, den z.B. für den Aufruf von CGIs zu benutzen.

Sonstiges

Bei Servern mit echt vielen offenen Deskriptoren kommt es zu dem lustigen Effekt, daß `open()` plötzlich meßbar länger dauert. Das liegt daran, daß `open()` laut POSIX den kleinsten verfügbaren Deskriptor wählen muß.

Das tut der Kernel, indem er ein Bitfeld linear durchgeht.

Es ist also theoretisch denkbar, daß man ein paar Nanosekunden sparen kann, wenn man die unteren Dateideskriptoren mit dup2 frei hält.