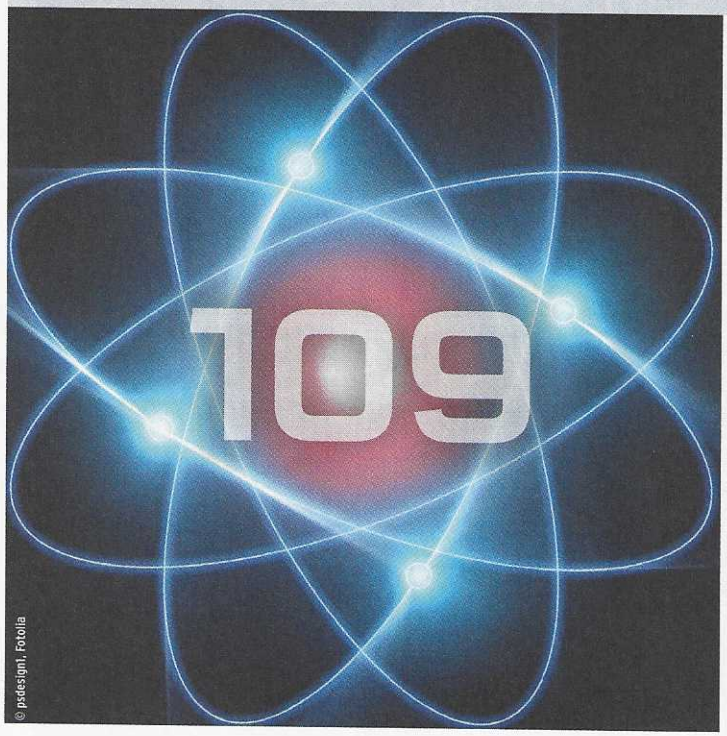
Kernel- und Treiberprogrammierung mit dem Linux-Kernel – Folge 109

Kern-Technik

Async-IO reloaded: Mit dem neuen Io_uring-Subsystem geben die Kernel-Entwickler Programmierern die Möglichkeit, die Performance von Datentransfers signifikant zu steigern. Anstelle von Funktionen stehen Datenstrukturen im Mittelpunkt. Eva-Katharina Kunst, Jürgen Quade



© psdesign1, Fotolia

Vor rund 50 Jahren haben die Unix-Schöpfer Ken Thompson und Dennis Ritchie ein Interface für den Zugriff auf Dateien und Peripherie (I/O) entworfen, das durch Eleganz, Usability und Effizienz besticht. Über »open«, »close«, »read« und »write« werden Daten – woher und wohin auch immer – gelesen, geschrieben oder per Pipe an andere Applikationen weitergeleitet.

Diese Art des Zugriffs bezeichnet man als synchrone Ein-/Ausgabe, weil die Anwendung synchron zum Programmablauf einen Datenblock nach dem anderen verarbeitet (Abbildung 1). Während die Applikation sich Block für Block vornimmt, drehen bei modernen Multicore-CPUs allerdings diverse Prozessorkerne Idle-Runden und bleiben ungenutzt. Dass Linux von sich aus versucht, die Schreib- und Leseaufträge unterschiedlicher Programme zu parallelisieren, nutzt der einzelnen Applikation dabei herzlich wenig.

In Form von Async-IO bietet der Torvaldsche Betriebssystemkern bereits seit Jahren ein asynchrones Interface. Applikationen spezifizieren einen Haufen Transferaufträge, übergeben sie dem Kernel und holen sich zu einem späteren Zeitpunkt die Ergebnisse ab [1]. Was so einfach und logisch klingt, ist für den Programmierer allerdings ein wahrer Alptraum. Bei aller Effizienz bleiben Eleganz und Usability auf der Strecke. Kein Wunder also, dass lediglich große Datenbanken das Interface nutzen und es ansonsten ein tristes Nischendasein fristet.

Zweiter Aufschlag

Kernel-Entwickler Jens Axboe hat jetzt mit Io_uring einen neuen Anlauf unternommen, der seit Version 5.1 im Kernel integriert ist und mit jeder neuen Version Optimierung und Erweiterungen erfährt. Sein Hauptaugenmerk hat Axboe dabei auf die Performance gelegt – mit der Konsequenz, dass Linux-Profis ihre Transferraten signifikant steigern können, Gelegenheitsprogrammierer aber weiterhin Zaungäste bleiben. Kein Interface für die Massen also.

Eleganz und Pfiffigkeit kann man Io_uring allerdings nicht absprechen. Axboe hat nämlich weniger eine Funktionsschnittstelle entworfen als vielmehr ein Daten-Interface. Anders ausgedrückt: Transferaufträge erfolgen nicht primär durch den Aufruf einer Funktion, sondern vielmehr durch das Beschreiben einer Datenstruktur. Dasselbe gilt für das Abholen der Ergebnisse. Auf diese Weise lassen sich aufwendige Kontextwechsel einsparen, und Linux braucht dank des datengetriebenen Ansatzes nur drei neue System-Calls. Die Konsequenz für den Programmierer: Sein Augenmerk richtet sich auf die Datenstrukturen.

Datenstrukturen statt Funktionen

Zum Verwalten der Ein-/Ausgabeaufträge gibt es drei Datenstrukturen: Eine Submission Queue Entry (»struct io_uring_sqe«) beschreibt die Aufträge selbst, die Completion Queue Entry (»struct io_uring_cqe«) informiert über die Abarbeitung, und die Struktur Params (»struct io_uring_params«) zeichnet für die generelle Verwaltung verantwortlich. Abbildung 2 zeigt die zentralen Datenstukturen von Io_uring.

Als Erstes spezifiziert eine Applikation die Anzahl der maximal parallel abzuarbeitenden Aufträge, also die Anzahl der benötigten SQEs. Dann fordert sie den Kernel per System-Call »io_uring_setup()«

Die Autoren

Eva-Katharina Kunst ist seit den Anfängen von Linux Fan von Open Source. Jürgen Quade, Professor an der Hochschule Niederrhein, bietet auch für Unternehmen Schulungen zu den Themen Treiberprogrammierung und Embedded Linux an.

auf, passende Datenstrukturen anzulegen. Der Kernel erledigt das und quittiert seine Aktion durch Ausfüllen einer Instanz der Datenstruktur »io_uring_params«.

Adressrechner für Kernel und Applikation

An dieser Stelle kommt es zu Komplikationen: Kernel und Applikation arbeiten zwar auf denselben Datenstrukturen (Shared Memory), haben aber eigene und damit unterschiedliche Adressräume. Daher muss der vom Kernel angelegte Speicher zunächst in den Adressraum der Applikation eingebunden (gemappt) werden.

Außerdem dürfen die Datenstrukturen im gemeinsamen Speicher keine absoluten Adressangaben enthalten, die ja nur für eine Seite (Kernel oder Applikation) gelten. Stattdessen verwenden sie nur Offsets zum Anfang des Speichers, anhand derer Kernel und Applikation die realen Adressen im eigenen Adressraum errechnen. Die Dokumentation zu Io_uring schlägt vor, diese Rechnung zu Beginn einmalig vorzunehmen, zu speichern und dann mit den gespeicherten absoluten Adressen weiterzuarbeiten (Abbildung 3, »app_sq_ring« und »app_cq_ring«).

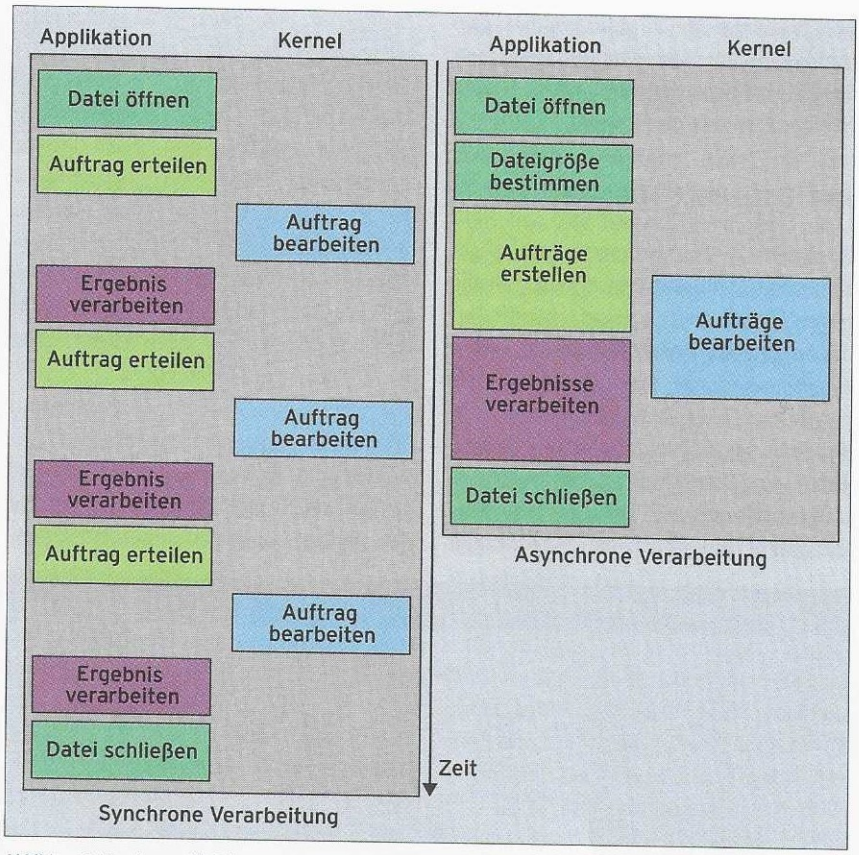


Abbildung 1: Synchrone (links) versus asynchrone Verarbeitung (rechts).

Vektor-IO

Um ein doppeltes Kopieren (von der Peripherie zum Kernel, vom Kernel zur Applikation) zu vermeiden, wie es beim synchronen Zugriff gang und gäbe ist, nutzen Betriebssystemkern und Applikation auch die Speicherbereiche für die eigentlichen Daten gemeinsam. Diese Bereiche werden im Gegensatz zu den Verwaltungsstrukturen als sogenanntes Vektor-IO (Iovec) durch die Applikation angelegt. Es liegt dann am Kernel, sie in seinen Adressraum einzublenden. Dazu muss die Speicheradresse aber zwangsläufig an einer Page-Adresse ausgerichtet sein.

Zur Erinnerung: Linux teilt seinen Speicher in Blöcke ein, die normalerweise jeweils 4096 Byte umfassen. Für ein Turbo-Copy muss sich zwangsläufig auch die Applikation an dieses Format halten. Dazu verwendet sie statt der klassischen Speicherreservierungsfunktion »malloc()«

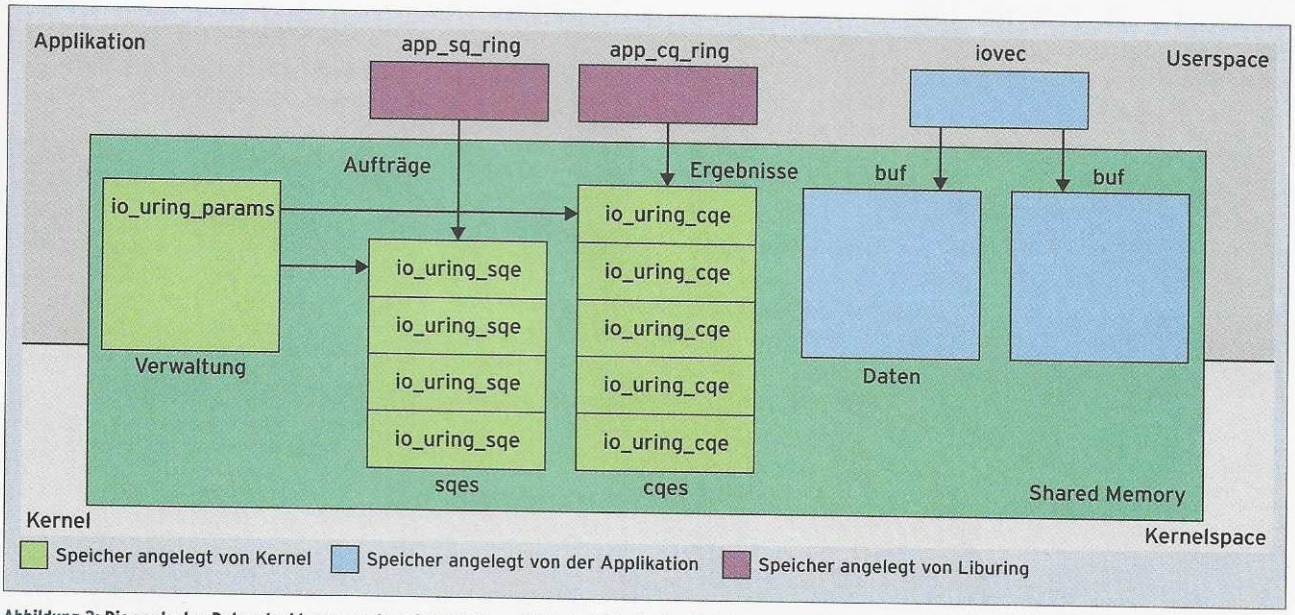


Abbildung 2: Die zentralen Datenstrukturen von Io_uring gliedern sich in drei Abschnitte. Dabei arbeiten Programm und Kernel auf denselben Datenstukturen.

beispielsweise die Funktion »posix_memalign()«. Die Adressen der fraglichen Speicherbereiche und deren Länge landen in einer Liste (»struct iovec«).

Auftragsvergabe

Nachdem diese Vorbereitungen abgeschlossen sind, kann es nun zur Sache gehen. Vereinfacht gesagt, spezifiziert die Applikation Transferaufträge in jeweils eigene Submission Qeue Entries, die sie per Index referenziert. Abbildung 4 verdeutlicht das zugrundeliegende Prinzip.

Jeden einzelnen Auftrag kennzeichnen unter anderem ein Opcode, ein File-Deskriptor, ein Offset, ein zugeordneter Applikationsspeicher, eine Längenangabe sowie ein frei definierbares Magic.

Der Opcode spezifiziert, was der Kernel überhaupt machen soll, beispielsweise lesen oder schreiben. Der File-Deskriptor kennzeichnet die beteiligte Datei oder das Peripheriegerät; der Offset gibt an, ab welcher Adresse der Datei beziehungsweise der Peripherie der Zugriff erfolgen soll. Die Datei (oder die Peripherie) wird also vor dem asynchronen Zugriff ganz normal per »open()« geöffnet.

Die Applikation stellt außerdem einen oder mehrere Speicherbereiche zur Verfügung, die als Quelle beziehungsweise als Ziel des Transfers dienen, inklusive der Anzahl zu transferierender Bytes. Bleibt zu guter Letzt noch das Magic, das eine Verbindung zwischen dem Auftrag und der späteren Meldung bezüglich dessen Erledigung herstellen soll.

Ein Ring, sie alle zu finden

Hat die Applikation auf diese Weise einen oder mehrere Aufträge spezifiziert, trägt sie die Indizes der entsprechenden SQEs in die Verwaltungsstruktur »sqe_ring« ein. Das wesentliche Element dieser Datenstruktur bildet der Ringpuffer, der dem ganzen Subsystem den Namen gab und den Kern des Mechanismus bildet.

Beim Ringpuffer handelt es sich um ein Array, dessen Länge ein Vielfaches von

Listing 1: Vereinfachtes Cat auf Basis von Liburing

```
001 #include <stdio.h>
002 #include <stdlib.h>
003 #include <fcntl.h>
004 #include <unistd.h>
005 #include <string.h>
006 #include <ctype.h>
007 #include <sys/stat.h>
008 #include <liburing.h>
009
010 static off_t get_file_size(int fd)
011 {
012   struct stat st;
013
014   if (fstat(fd, &st) == 0)
015     if (S_ISREG(st.st_mode))
016       return st.st_size;
017   return 0;
018 }
019
020 int main( int argc, char **argv, char **envp
    )
021 {
022   struct io_uring ring;
023   struct io_uring_sqe *sqe;
024   struct io_uring_cqe *cqe;
025   struct iovec *iovecs;
026   int fd, i, ret;
027   off_t filesize;
028   off_t offset;
029   void *buf;
030   char *ptr;
031   int entries;
032
033   if (argc!=2) {
034     fprintf(stderr,"usage: %s filename\n",
    argv[0]);
035     return -1;
036   }
037   fd = open( argv[1], O_RDONLY );
038   if (fd<0) {
039     perror( argv[1] );
040     return -1;
041   }
042   filesize = get_file_size( fd );
043   if (filesize==0 || filesize>(4096*64)) {
044     fprintf(stderr,"%s: file invalid or too
    big\n",argv[1]);
045     return -1;
046   }
047
048   // Verwaltungsstrukturen anlegen lassen
049   entries = (filesize+4095) / 4096;
050   ret = io_uring_queue_init(entries, &ring,
    0);
051   if (ret<0) {
052     fprintf(stderr,"queue_init:
    %s\n",strerror(-ret));
053     return -1;
054   }
055
056   // Memory fuer die Daten reservieren
057   iovecs = calloc( entries, sizeof(struct
    iovec));
058   if (iovecs==NULL) {
059     perror( "calloc" );
060     return -1;
061   }
062   // Memory allocation
063   for (i=0; i<entries; i++) {
064     if (posix_memalign(&buf, 4096, 4096))
065       return 1;
066     iovecs[i].iov_base = buf;
067     iovecs[i].iov_len = 4096;
068   }
069
070   // Lese-Auftraege vorbereiten
071   offset = 0;
072   for (i=0;i<entries;i++) {
073     sqe = io_uring_get_sqe(&ring);
074     if (sqe==NULL)
075       break;
076     io_uring_prep_readv(sqe,fd,&iovecs[i],1
    ,offset);
077     sqe->user_data = (unsigned long)
    iovecs[i].iov_base;
078     offset += iovecs[i].iov_len;
079   }
080
081   // Auftraege uebergeben und auf
    Fertigstellung warten
082   ret = io_uring_submit_and_wait(&ring,
    entries);
083   if (ret < 0) {
084     fprintf(stderr, "io_uring_submit_and_
    wait: %s\n",
085       strerror(-ret));
086     return 1;
087   }
088   // Ergebnisse verarbeiten
089   while (io_uring_peek_cqe(&ring,&cqe)==0
    ) {
090     fprintf(stderr,"completion %d - magic:
    %llx\n",
091       cqe->res, cqe->user_data);
092     ptr = (char *)cqe->user_data;
093     for (i=0;i<cqe->res;i++) {
094       printf("%c", isascii(*(ptr+i))?*(pt
    r+i):'.');
095     }
096     io_uring_cqe_seen(&ring, cqe); //
    Quittieren
097   }
098   // Ressourcen freigeben
099   io_uring_queue_exit(&ring);
100   free( iovecs );
101   close( fd );
102   return 0;
103 }
```

zwei Elementen beträgt, beispielsweise 8, 16 oder 32. Es gibt die beiden Zeiger »tail« und »head«. Die Applikation schreibt den Index des ausgefüllten SQEs in das Element im Array, auf das der Tail-Zeiger verweist; danach wird er inkrementiert. Dabei darf der Tail-Zeiger den Head-Zeiger nicht überholen. Tut er das doch (»tail > head«), dann ist der Ringpuffer voll, und er kann kein weiteres Element aufnehmen. In diesem Fall muss die Applikation auf einen freien Slot warten. Umgekehrt signalisiert »tail == head« einen leeren Ringpuffer.

Initial muss die Applikation dem Kernel-Subsystem einmal mitteilen, dass der Kernel anfangen soll, die Aufträge zu bearbeiten. Einmal in Gang gesetzt, wird allerdings nur der Tail-Zeiger inkrementiert, damit der Kernel sich den nächsten Auftrag vorknöpft. Sobald er einen Auftrag akzeptiert, erhöht er den Head-Zeiger und signalisiert damit zugleich der Applikation, dass der SQE erneut eingesetzt werden kann.

Der Kernel gibt also den SQE frei, obwohl der Auftrag möglicherweise noch gar nicht beendet wurde. Er speichert alle notwendigen Informationen zwischen. Das betrifft freilich nur den SQE, nicht aber den hinter dem Auftrag liegenden Datenbereich. Der lässt sich erst freigeben respektive erneut einsetzen, wenn der Kernel über den CQE-Ring die Erledigung meldet.

Ruhe sanft

Die Applikation hat zwei Möglichkeiten, das zu erkennen: Entweder überwacht sie die Tail- und Head-Zeiger des CQE-Rings,

Liburing-Installation

Zurzeit fällt noch Handarbeit an, um die von Jens Axboe geschriebene Liburing nutzen zu können. Grundvoraussetzung ist zunächst ein Linux-Kernel ab Version 5.1, den man nur auf topaktuellen Systemen findet.

Wer unter Ubuntu 18.04 das Paket »linux-generic-hwe-18.04« installiert, bekommt jedoch einen geeigneten Kernel (momentan in der Version 5.3). Die Bibliothek selbst gibt es gegenwärtig nur im Quellcode, den der Programmierer selbst generieren und installieren muss. Unter Ubuntu 18.04 bereitet das aber erfreulicherweise keinerlei Probleme (Listing 2).

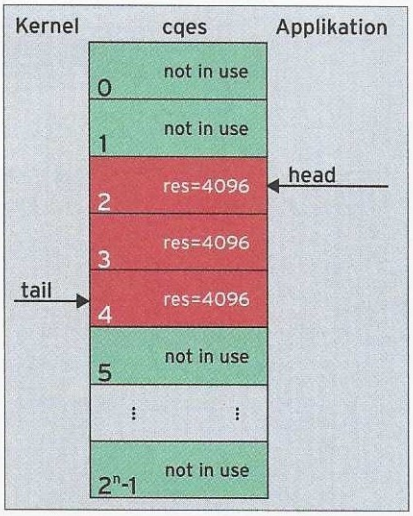


Abbildung 3: Der Ringpuffer hat dem Subsystem seinen Namen gegeben.

oder sie legt sich mithilfe des Systemaufrufs »io_uring_enter()« bis zur Erledigung schlafen (Abbildung 3). Da die SQEs bereits vor Erledigung des Auftrags der Applikation wieder zur Verfügung stehen, kann es zu einem Backlog bei den Erledigungsnachrichten (Completions) kommen. Aus diesem Grund hält der Kernel doppelt so viele CQEs wie SQEs vor.

Um sich bis zum Ende der Bearbeitung schlafen zu legen, verwendet die Applikation den System-Call »io_uring_enter()«. Problematisch ist noch die Zuordnung eines Completion-Queue-Events zum eigentlichen Auftrag. Genau das soll das bereits erwähnte Magic »user_data« gewährleisten. Der Kernel kopiert es vom ursprünglichen Auftrag in die Completion-Struktur. Im einfachsten Fall trägt die Applikation hier die Adresse des Applikationsspeichers ein.

Neben dem Magic enthält die Datenstruktur »struct io_uring_cqe« am Ende noch die Anzahl der zu transferierenden Bytes.

Take it easy

Implementiert Liburing eine elegante Technik? Ja. Ist es leistungsfähig? Zweifellos. Lässt die Technik sich einfach handhaben? Fehlanzeige. Die Applikation muss asynchron programmiert werden (Abbildung 1), hinzu kommt die Systemintegration mit beispielsweise »mmap()« und Iovecs. Das wissen auch die Linux-Entwickler, und Jens Axboe hat mit der Bibliothek Liburing zumindest die Programmierung etwas erleichtert.

Zum Setup genügt es, die Datenstruktur »struct io_uring ring« zu definieren und die Funktion »io_uring_queue_init(entries, &ring, 0)« aufzurufen (Listing 1, Zeile 51). Die Liburing verbirgt dabei für den Programmierer das gesamte Adressraum-Mapping. Die Iovecs, die die zu transferierenden Daten aufnehmen, muss er aber weiterhin zum Beispiel durch Aufruf der Funktion »posix_memalign()« in der Applikation bereitstellen (Zeile 65).

Für den eigentlichen Datentransfer lässt sich die Applikation erst einmal durch Aufruf der Funktion »io_uring_get_sqe()« den nächsten freien Auftragsblock zuweisen (Zeile 74). Mithilfe des Makros »io_uring_pre_rw()« füllt sie dann den Auftragsblock aus.

Nach dem Ausfüllen mindestens eines Auftrags kann der eigentliche Transfer durch Aufruf von »io_uring_submit()« gestartet werden. In Listing 1 kombiniert die Funktion »io_uring_submit_and_wait()« (Zeile 83) das Abschicken der Aufträge und das Schlafen bis zur deren kompletter Erledigung.

Ergebnisliste

Es gibt auch Funktionen, die beim Verwalten von Erledigungsmeldungen durch den Kernel helfen. Der Beispielcode aus Listing 1 holt diese Completions einzeln per »io_uring_peek_cqe()« ab.

Um den Code übersichtlich zu halten, geht das Beispiel jedoch von der in der Praxis nicht zutreffenden Annahme aus, dass der Kernel die Aufträge sequenziell abarbeitet. Wegen der Parallelität innerhalb des Kernels können sich die Aufträge aber tatsächlich ohne Weiteres auch gegenseitig überholen. Dadurch kann die Ausgabe in seltenen Fällen ein wenig durcheinander geraten.

Hat die Applikation eine Completion-Meldung verarbeitet, dann teilt sie das

Listing 2: Generierung und Installation der Liburing

```
01 $ sudo apt-get install --install-recommends
   linux-generic-hwe-18.04
02 $ git clone https://github.com/axboe/liburing.git
03 $ cd liburing
04 $ ./configure
05 $ make
06 $ sudo make install
```

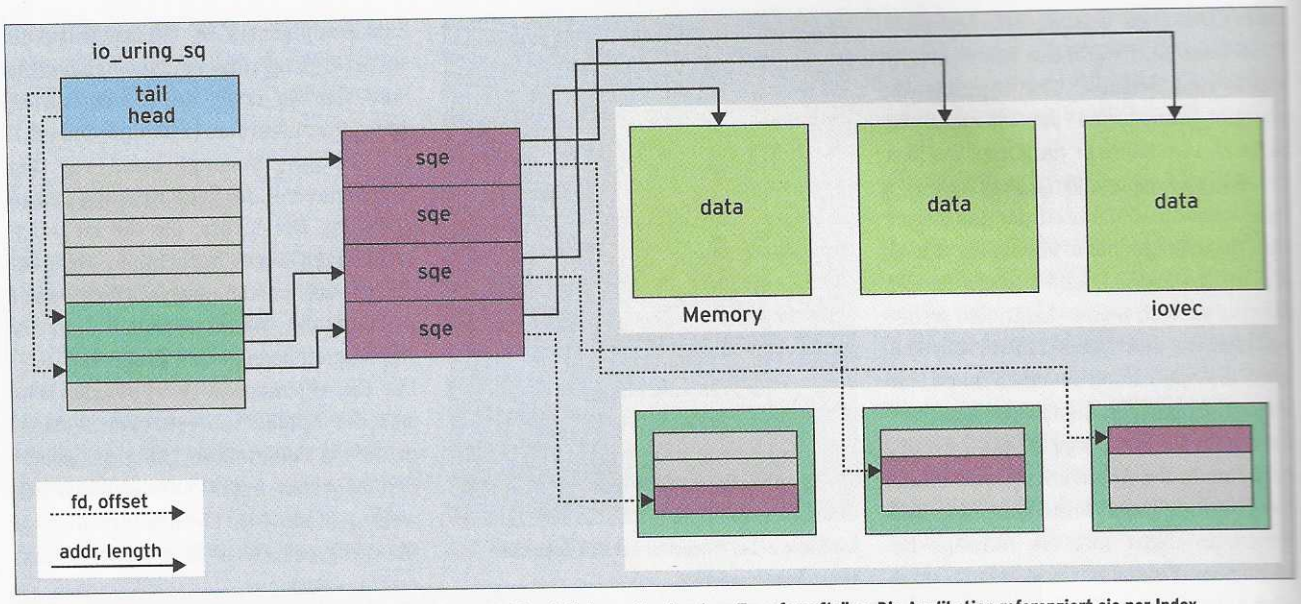


Abbildung 4: Von der Applikation erstellte Submission Queue Entries definieren die einzelnen Transferaufträge. Die Applikation referenziert sie per Index.

dem Kernel durch Aufruf von »io_uring_cqe_seen()« mit. Nach Abschluss des gesamten Transfers gibt sie per »io_uring_queue_exit()« vom Kernel reservierte Ressourcen wieder frei.

Eigene Gehversuche mit dem Io_uring-Subsystem

Der Code aus Listing 1 lässt sich für erste eigene Gehversuche mit Io_uring und Liburing nutzen. Es handelt sich dabei um eine stark vereinfachte Version des Kommandozeilen-Tools Cat, die die als Parameter mitgegebene Datei auf der Konsole ausgibt.

Haben Sie die Quellcode-Datei als »mycat.c« abgespeichert, Liburing auf Ihrem Rechner installiert (siehe Kasten „Liburing-Installation") und läuft bei Ihnen ein Kernel in der Version 5.1 oder höher, dann generieren Sie das Executable »mycat« durch den folgenden Aufruf:

```
$ make CFLAGS="-D_GNU_SOURCE" LDLIBS=➥
-luring mycat
```

Starten Sie das Programm dann mit »./mycat mycat.c«, erscheint der Quellcode – eventuell etwas verwürfelt – auf dem Bildschirm.

Die Liburing bringt weiteren Beispielcode mit. Den gilt es gründlich zu studieren, um erfolgreiche komplexere Szenarien zu programmieren: Aufgrund der zurzeit noch mangelhaften Dokumentation [2] muss man sich das benötigte Know-how via Quellcodestudium erschließen. Das ist sehr schade, denn Io_uring hat weit mehr zu bieten [3], als hier angesprochen wurde [4]. Der Blick auf die Opcodes, also die implementierten Transferkommandos, zeigt aktuell knapp 30 unterschiedliche Varianten, beispielsweise für den Datentransfer übers Netzwerk oder die Kombination mit Timeout-Zeiten.

Bleibt zum Schluss noch die Frage, um wie viel Io_uring die Zugriffe überhaupt beschleunigt? Erste Anhaltspunkte dazu liefert ein Vortrag, den Entwickler Jens Axboe im Oktober vergangenen Jahres auf der Linux-Kernel-Konferenz Kernel Recipes 2019 in Paris gehalten hat [5]. Ohne ins Detail der Tests zu gehen, punktet demzufolge der neue Ansatz im Vergleich zum alten Async-IO mit einer zwei- bis vierfachen Geschwindigkeit. Bereits auf einem Single-Core-System wickelt der neue Ansatz das Lesen einer 16 GByte großen Datei in doppelter Geschwindigkeit ab. Das ist mehr als beeindruckend – und für Applikationen mit großen Datenbewegungen eine ausreichende Begründung, sich näher mit Io_uring auseinanderzusetzen. (*jlu*) ■

Tabelle 1: Ausgewählte Funktionen der Liburing

Funktion	Aufgabe
»io_uring_queue_init()«	Anlegen der Verwaltungsstrukturen durch den Kernel
»io_uring_queue_exit()«	Freigabe reservierter Ressourcen
»io_uring_get_sqe()«	Anfordern eines freien Auftragsblocks
»io_uring_prep_rw()«	Auftrag erstellen
»io_uring_prep_readv()«	Leseauftrag (Vektor-IO) erstellen
»io_uring_prep_writev()«	Schreibauftrag (Vektor-IO) erstellen
»io_uring_submit()«	Auftrag zur Abarbeitung übergeben
»io_uring_submit_and_wait()«	Aufträge übergeben und auf Abarbeitung warten
»io_uring_peek_cqe()«	Ergebnis abholen
»io_uring_wait_cqe_nr()«	auf Abarbeitung warten
»io_uring_cqe_seen()«	Ergebnisstruktur zur erneuten Nutzung freigeben

Infos

[1] Async-I/O: Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik", LM 02/2005, S. 86, [https://www.linux-magazin.de/7282]
[2] „Efficient IO with io_uring": [https://kernel.dk/io_uring.pdf]
[3] „Ringing in a new asynchronous I/O API": [https://lwn.net/Articles/776703/]
[4] „The rapid growth of io_uring": [https://lwn.net/Articles/810414/]
[5] „Faster IO through io_uring": [https://kernel-recipes.org/en/2019/talks/faster-io-through-io_uring/]