Linux – Prinzipien und Programmierung

Dr. Klaus Höppner

Hochschule Darmstadt – Wintersemester 2011/2012

Verwenden von Puffern

Sockets sind relativ normale Filedeskriptoren (mit Einschränken, so ist eine absolute Positionierung natürlich nicht möglich). Analog zu Pipes können Sockets als gepuffert werden.

Hierbei gilt als Faustregel, dass ein Socket niemals mit einem `FILE` gleichzeitig zum Lesen und Schreiben gepuffert werden darf. Daher ist folgender Code typisch:

```
FILE* in = fdopen(sock, "r");
FILE* out = fdopen(dup(sock), "w");
```

Beispiel-Client

Im folgenden Beispiel wird meine Webseite über eine gepufferte Verbindung angefordert und gelesen (und nebenbei demonstriert, wie Namen per DNS aufgelöst werden können):

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <stdio.h>

#define MAX 100

int main() {
    char * hostname = strdup("www.klaus-hoeppner.de");
    struct hostent * host = gethostbyname(hostname);
```

Beispiel-Client (Forts.)

```
if (host==NULL) {
    printf("Kann host nicht aufloesen\n");
    exit(EXIT_FAILURE);
}

struct in_addr * host_addr = (struct in_addr *) host->h_addr_list[0];
printf("IP: %s\n", inet_ntoa(*host_addr));

int sock = socket(AF_INET, SOCK_STREAM, 0);
if (sock<0) {
    printf("Fehler %d beim socket-Erzeugen: %s\n",
           errno, strerror(errno));
    exit(EXIT_FAILURE);
}

struct sockaddr_in address;
memset(&address, 0, sizeof(address));
address.sin_family = AF_INET;
address.sin_port = htons(80);
address.sin_addr = *host_addr;
```

Beispiel-Client (Forts.)

```
    if (connect(sock, (struct sockaddr *) &address, sizeof(address))<0) {
        printf("Fehler %d beim connect: %s\n",
               errno, strerror(errno));
        exit(EXIT_FAILURE);
    }

    FILE* in = fdopen(sock, "r");
    FILE* out = fdopen(dup(sock), "w");
    fprintf(out, "GET / HTTP/1.1\r\nHost: %s\r\n\r\n", hostname);
    fflush(out);

    char buf[MAX];
    while (1) {
        if (fgets(buf, MAX-1, in)==NULL) break;
        printf(buf);
    }
    close(sock);
    return(EXIT_SUCCESS);
}
```

Parallele Server

Der bisherige Echo-Server kann nur einen Client gleichzeitig bedienen, d. h. während ein Client verbunden ist, müssen alle anderen warten.

Nun wird der Echo-Server so modifiziert, dass er für jeden von max. 5 Clients einen neuen Serverprozess per `fork` erzeugt. Dies entspricht z. B. dem üblichen Vorgehen beim Apache-HTTPD (wo zur Beschleunigung meist schon per *prefork* mehrere Server im Voraus angelegt werden).

Änderung an der Implementierung

```
void child_exited(int signum) {
    int pid, status;
    while ( (pid=wait(&status)) != -1) {
        printf("Kindprozess %d beendet\n", pid);
    }
}

// Änderungen in main()

listen(sock, 5);
sethandler(SIGCHLD, child_exited);

char buf[MAX];
struct sockaddr_in client_address;
size_t addrlen = sizeof(client_address);
while(1) {
    int client_sock = accept(sock,
            (struct sockaddr *) &client_address, &addrlen);
    if (client_sock<0) {
        printf("Fehler %d beim accept: %s\n",
               errno, strerror(errno));
        exit(EXIT_FAILURE);
    }
```

Änderung an der Implementierung

```
    int child_pid = fork();
    if (child_pid==0) {
        close(sock);
        printf("Verbindung von %s, Port %d\n",
               inet_ntoa(client_address.sin_addr),
               ntohs(client_address.sin_port));
        sprintf(buf, "Bereit fuer Echo (quit fuer Ende)\n");
        send(client_sock, buf, strlen(buf), 0);
        do {
            int size = recv(client_sock, buf, MAX-1, 0);
            buf[size] = '\0';
            printf("%d Zeichen gelesen: +++%s+++\n", size, buf);
            send(client_sock, buf, strlen(buf), 0);
        } while (strcmp(buf,"quit"));
        close(client_sock);
        exit(EXIT_SUCCESS);
    }
    close(client_sock);
}
```

Verbindungslose Kommunikation mit UDP

Bei der verbindungslosen Kommunikation per UDP schickt ein Client Pakete an einen Server (oder an alle), ohne dass über Handshakes der Erfolg der Übertragung sicher gestellt ist.

Da so ein großer Teil des Overheads wegfällt, ist die Kommunikation so deutlich Ressourcen-schonender. Der Nachteil liegt natürlich darin, dass man sich auf die Kommunikation nicht verlassen kann. D. h. eine Anwendung muss so implementiert sein, dass sie robust dagegen ist, dass eine verschickte Nachricht nicht ankommt.

Beispiel

Im folgenden Beispiel sollen ein oder mehrere Clients verbindungslos den Namen und die Position eines Spielers an einen Server übertragen.

Hierfür wird in `position.h` folgende Struktur definiert:

```
#ifndef _POSITION_H
#define _POSITION_H

typedef struct {
    char name[40];
    int pos;
} position_t;

#endif
```

Implementierung Server

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <stdio.h>
#include "position.h"

#define MAX 1024

int main() {
    int sock = socket(AF_INET, SOCK_DGRAM, 0);

    struct sockaddr_in address;
    memset(&address, 0, sizeof(address));
    address.sin_family = AF_INET;
    address.sin_port = htons(9999);
    address.sin_addr.s_addr = htonl(INADDR_ANY);

    bind(sock, (struct sockaddr *) &address, sizeof(address));
```

Implementierung Server (Forts.)

```
    position_t pos;
    struct sockaddr_in client_address;
    size_t addrlen = sizeof(client_address);
    while(1) {
        recvfrom(sock, &pos, sizeof(pos),
                 0, (struct sockaddr *) &client_address, &addrlen);
        printf("Daten von %s, Port %d, Name %s, Position %d\n",
               inet_ntoa(client_address.sin_addr),
               ntohs(client_address.sin_port),
               pos.name, pos.pos);
    }

    return(EXIT_SUCCESS);
}
```

Implementierung Client

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <stdio.h>
#include "position.h"

#define MAX 1024

int main(int argc, char** argv) {
    int sock = socket(AF_INET, SOCK_DGRAM, 0);

    struct sockaddr_in client_address;
    memset(&client_address, 0, sizeof(client_address));
    client_address.sin_family = AF_INET;
    client_address.sin_port = htons(0);
    client_address.sin_addr.s_addr = htonl(INADDR_ANY);

    bind(sock, (struct sockaddr*) &client_address, sizeof(client_address))
```

Implementierung Client (Forts.)

```
    struct sockaddr_in address;
    memset(&address, 0, sizeof(address));
    address.sin_family = AF_INET;
    address.sin_port = htons(9999);
    inet_aton("127.0.0.1", &address.sin_addr);

    position_t pos;
    strcpy(pos.name, argv[1]);
    int x;
    for (x=0; x<10; x++) {
        pos.pos = x;
        sendto(sock, &pos, sizeof(pos),
               0, (struct sockaddr*) &address, sizeof(address));
        sleep(1);
    }
    close(sock);

    return(EXIT_SUCCESS);
}
```

Unix Domain Sockets

Ein Socket muss nicht zwingend eine Kommunikation über Netzwerk beschreiben.

Eine bereits lange verbreitete Methode besteht in dem Verwenden von *Unix Domain Sockets*, bei denen die Kommunikation über eine Datei stattfindet. Dies kann also auf einem Rechner zur Inter-Prozess-Kommunikation benutzt werden.

Der Umstieg von IP-Kommunikation zu Socket-Files ist einfach:

- Änderung der Familie von `AF_INET` nach `AF_UNIX`
- Aufbau einer Struktur des Typs `sockaddr_un` statt `sockaddr_un`.

Grundgerüst Server

```
#include <sys/socket.h>
#include <sys/un.h>
...
int main() {
    int sock = socket(AF_UNIX, SOCK_STREAM, 0);
    struct sockaddr_un address;
    memset(&address, 0, sizeof(address));
    address.sun_family = AF_UNIX;
    strcpy(address.sun_path, "/tmp/myecho.sock");

    bind(sock, (struct sockaddr *) &address, sizeof(address));
    listen(sock, 1);

    struct sockaddr_un client_address;
    size_t addrlen = sizeof(client_address);
    while(1) {
        int client_sock = accept(sock,
                (struct sockaddr *) &client_address, &addrlen);
        ...
        close(client_sock);
    }
    return(EXIT_SUCCESS);
}
```

Grundgerüst Client

```
#include <sys/socket.h>
#include <sys/un.h>
...
int main() {
    int sock = socket(AF_UNIX, SOCK_STREAM, 0);
    struct sockaddr_un address;
    memset(&address, 0, sizeof(address));
    address.sun_family = AF_UNIX;
    strcpy(address.sun_path, "/tmp/myecho.sock");

    connect(sock, (struct sockaddr *) &address, sizeof(address));

    ...

    close(sock);

    return(EXIT_SUCCESS);
}
```

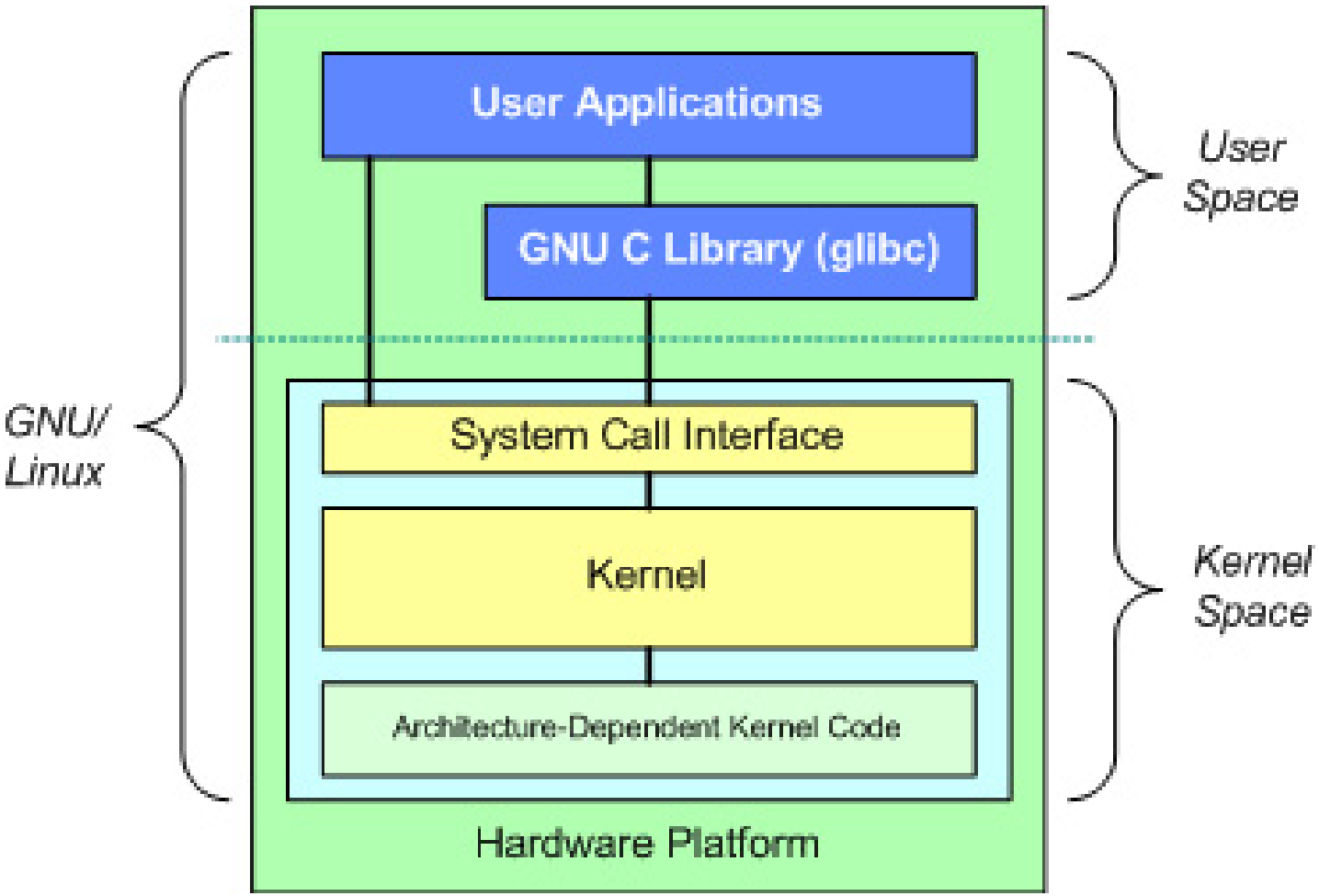
Aufgabe des Kernels

Herzstück des Betriebssystems

- Verwaltung und Bereitstellung von Hardware und Systemressourcen
- Typische Komponenten:
  Interrupt-Handler, Process Scheduler, Memory Management, System Services (Networking, IPC)
- Unterscheidung: User Space $\leftrightarrow$ Kernel Space

Grundlegendes Schema:
Eine Anwendung wird im User Space in einem Prozess ausgeführt, macht einen *System Call* (z. B. `printf`), dieser wird im Kernel Space im Prozess-Kontext ausgeführt.
Zusätzlich laufen im Kernel Space Aktionen ab, die nicht im Kontext eines Prozesses liegen, z. B. Behandlung von Interrupts.

Schaubild


User Applications
GNU C Library (glibc)
System Call Interface
Kernel
Architecture-Dependent Kernel Code
Hardware Platform
User Space
Kernel Space
GNU/ Linux

Monolithischer Kernel

Der Linux-Kernel ist ein monolithischer Kernel.
Bedeutet: Pinzipiell kann der Kernel ein einzelnes Executable sein, das in einem einzelnen Prozess mit globalem Adressraum läuft.

Vorteil:	einfacher zu implementieren, ressourcen-schonend (keine IPC nötig)
Nachteil:	Kein Schutzmechanismus, Kernel Code hat freien Zugriff auf den gesamten Adressraum

Besonderheit bei Linux: Zwar monolithisch, aber modular, d. h. der Kernel hat die Möglichkeit, Kernel Code dynamisch hinzu zu laden oder zu entfernen.

Microkernel

Bei Microkerneln werden die Aufgaben des Kernels von einzelnen Prozessen ausgeführt, so genannten Servern.

Vorteil: Jeder Server braucht nur die Rechte, die für seine Aufgabe notwendig sind; getrennte Adressräume

Nachteil: Mehr Aufwand bei Implementation; IPC notwendig, das kostet Zeit.

Beispiele: Minix, GNU/Hurd

Zwitter: Windows ab NT (Hybridkernel)

Versions-Schema

Jede Linux-Kernelversion hat eine Nummer, bestehend aus

- Major Version
- Minor Version
- Revision
- optional Stable Release-Nummer (seit 2005)

Aktuelle Nummer: Ab 2012 wird die Major Version 3 verwendet, aktuell 3.1.8 (letzte stabile Version der Linie 2.6: 2.6.39.4

Hinweis: Die frühere Unterscheidung (gerade Minor-Nr. Produktionsversion, ungerade Entwicklerversion) wurde inzwischen aufgegeben.