

1. Begriffe

Benutzerapplikation Dies ist eine Anwendung, die direkt der Benutzer bedienen kann und auf die er Zugriff hat.

Betriebssystem Das Betriebssystem ist die Basissoftware, die den Betrieb eines Computers sichert. Es übernimmt so z.B. für den Benutzer die Speicherverwaltung, kontrolliert die Peripherie und steuert die Prozesse.

Systemaufruf Durch Systemaufrufe können Benutzerapplikationen auf Systemdienste zugreifen, die im System Interface liegen.

Time Sharing System Ermöglicht den Dialogbetrieb von mehreren Benutzern. Der Prozessor wird dabei periodisch unterbrochen und Programmen anderer Benutzer zugeteilt (Mehrprogrammbetrieb); für jedes Programm selbst sieht es jedoch so aus, als ob es seinen eigenen Prozessor hätte.

Microkernel Das ist der Betriebssystemkern, in dem in der Regel nur die Funktionalität der Speicher- und Prozessverwaltung eingebunden ist, sowie Grundfunktionen der Synchronisation und Kommunikation. Alle anderen Prozesse laufen im Userspace.

Monolithischer Kernel Im Gegensatz zum Microkernel werden hier mehr Funktionalitäten in den Kern implementiert. So z.B. die Treiber für Hardware o.ä.

Instruction Set Dies bezeichnet einen Satz von arithmetischen und logischen Befehlen sowie unter anderem load-store Anweisungen etc. der Prozessorarchitektur auf Hardware-Ebene. Der Umfang der Befehle variiert sehr stark zwischen den einzelnen Architekturen.

Interrupt Ein Interrupt ist ein Mechanismus, der das Unterbrechen des normalen CPU Fortlaufs durch eine Systemkomponente (Mouse, Tastatur, Speicher usw.) ermöglicht. Das Ziel hiervon ist natürlich die CPU Auslastung und damit die Effizienz zu verbessern.

2. Entwicklung von Betriebssystemen

- Ab 1963 wurde an einem neuen Betriebssystem namens Multics gearbeitet, das von MIT, General Electric, Bell Labs und Honeywell getragen wurde. Schnell wurde klar, dass das Projekt nicht tragbar sei, dennoch gab das Team nicht auf. Sie wollten sich ein Mehrbenutzersystem schaffen, das es ihnen erlaubte eine echte Arbeitsgemeinschaft zu bilden.
- Einige Prototypen eines Dateisystems und eines primitiven Kernels wurden entwickelt, die auf einer GE-645 lauffähig waren. Das System wurde spöttisch "Unics", in Anspielung

auf Multics, genannt, da es nur bis zu 2 Benutzer unterstützte. Später wurde er dann zu Unix verkürzt.

- 1972 begann die Neuerstellung von Unix in C, um zukünftig die Portierung von Unix auf neue Rechner zu erleichtern. Die Portierung wurde 1973 abgeschlossen und erhielt den Namen Unix V4. Im Jahr 1978 wurden bereits über 600 Computer mit UNIX-Betriebssystemen betrieben.
- 1979 wird schließlich durch AT&T die letzte UNIX-Version mit freiem Quellcode, nämlich UNIX V7, veröffentlicht. Microsoft erwirbt 1979 eine Unixlizenz und beginnt unter dem Namen Xenix die Arbeiten an Portierungen auf u. a. Intel 8086, Motorola 68000 und Zilog Z8000-Prozessoren.
- 1980 erschien die erste Portierung von UNIX V7 auf eine 32-Bit-Maschine. AT&T betrat offiziell den Computermarkt und begann 1983 ein auf Unix V7 basierendes System, zu vermarkten, während die Universität von Berkeley zeitgleich 4.2BSD veröffentlichte, das Neuerungen wie TCP/IP oder Signale mit sich brachte und somit eine Netzwerkschnittstelle bot.
- 1983 begann Richard Stallman mit der Arbeit an einem eigenen, Unix-ähnlichen Betriebssystem namens GNU und rief mit der Veröffentlichung des GNU Manifests[3] 1985 eine immer stärker werdende Bewegung für freie Software ins Leben.
- Das POSIX Standardisierungsprojekt wurde ins Leben gerufen, das eine einheitliche Schnittstelle für Unix definieren sollte. 1988 wurde schließlich POSIX.1 veröffentlicht (heute auch ein IEEE-Standard unter der Nummer 1003.1).
- 1987 entwickelte Andrew S. Tanenbaum ein Unix-ähnliches Betriebssystem namens Minix.
- Am 5. Oktober 1991 stellt Linus Torvalds seinen Kernel Linux mit der Versionsnummer 0.02 vor.
- 1992 Billy Jolitz veröffentlicht 386BSD, eine Portierung von 4.3BSD NET/2 auf den Intel 386er.
- 1993 4.4BSD erscheint und es beginnt die Entwicklung von FreeBSD und NetBSD basierend auf 386BSD.
- Der "Single UNIX Spec/2" Standard wird 1997 veröffentlicht, der jetzt Echtzeit, Threads und 64-Bit-Prozessoren unterstützt.
- 2000 Darwin, Unterbau des Mac OS X mit Mach als Kernel.
- 2002 ISO/IEC 9945:2002 (Single Unix Spec)

3. Protection Rings

Die Protection Rings stellen eine hierarchische Struktur für Prozesse dar. Je nach dem in welchem Ring der Prozess läuft, wird der Zugriff auf die CPU und andere Prozesse eingeschränkt bzw. erlaubt. Durch diese Ringe können Prozesse gut voneinander getrennt und abstrahiert werden, so wird auch eine gewisse Sicherheit gewährleistet, dass unprivilegierte Prozesse z.B. nicht in privilegierte Prozesse reinkommen können.

Normalerweise unterteilt man in 4 Ringe (Ring 0 bis Ring 3), wobei Ring 0 der mit der höchsten Priorität ist. Prozesse, die hier laufen, laufen im sogenannten "Kernel-Mode". In allen anderen Ringen spricht man vom "User-Mode". Im Ring 0 läuft natürlich dann auch der Kernel des Betriebssystems. In Ring 1 und 2 befinden sich die Systemdienste wie z.B. Treiber, Bibliotheken etc., im Ring 3 laufen schlussendlich die Benutzerapplikationen.

Die verbreiteten Betriebssysteme für x86 (dazu gehören Linux und Windows) nutzen jedoch nur zwei der vier möglichen Ringe. Im Ring 0 werden der Kernel und alle Hardwaretreiber ausgeführt, während die Anwendungssoftware im unprivilegierten Ring 3 arbeitet.

4. Fifty/Fifty

- *Ein Microkernel beschränkt sich auf grundlegende Speicherverwaltung und Kommunikation zwischen Prozessen.*

Ja, der Microkernel ist auf die genannten Funktionen beschränkt, sofern man Synchronisation noch unter Kommunikation zusammenfasst. Und zwar ist die beschränkte Funktionalität ja gerade das, was den Unterschied zu dem monolithischen Kernel ausmacht, da dort zusätzlich noch die Treiber und andere Dienste im Kernel integriert sind.

- *Im Kernelmodus muss das Betriebssystem darauf achten, dass die Speicherbereiche verschiedener Prozesse voneinander isoliert sind.*

Richtig. Wenn ein Prozess nämlich im Kernelmode läuft, hat er uneingeschränkten Zugriff auf alle möglichen Ressourcen, und somit ohne nötige Vorkehrungen auch auf andere Prozesse und deren Speicherbereiche.

- *Wenn Interrupt Service Routinen nebenläufig abgearbeitet werden, muss das Betriebssystem darauf achten, dass der Prozessor beim Timer-Interrupt nicht in den Benutzermodus wechselt.*

Nein. Nebenläufig bedeutet, dass Prozesse in beliebiger Reihenfolge ablaufen können. Bei dem Timer-Interrupt ist es jedoch wichtig, exakt zu bleiben, da dieser in bestimmten Intervallen ausgelöst wird, um die Systemzeit zu messen. Und wenn die ISR nebenläufig geht, dann funktioniert das nicht so ganz ...

- *Bei Microkernen kommt es häufiger zu einem Kontextwechsel als bei monolithischen Kernen.*

Diese Aussage ist wahr. Und zwar liegt das daran, dass nur minimale Funktionalitäten im Microkernel integriert sind und alle anderen Komponenten, so auch die Treiber,

im User Mode laufen und so über Systemcalls und Interrupts Prozessorzeit einfordern müssen. Im monolithischen Kernel hingegen laufen alle Betriebssystemfunktionen im Kernel Mode und somit entfällt das hier aufkommende Kontextwechseln.

5. Syscalls

1. Die erste Variante ist, dass einfach eine Subroutine aufgerufen wird. Das findet nur in einfach gestrickten OS ohne separate Adressräume Verwendung. Hier wird direkt durch den Aufruf der gewünschten Adresse in den Kernelmode gesprungen. Und genauso direkt wird per "return" wieder an die gleiche Stelle im Programm zurückgekehrt.
2. In der unter UNIX geläufigen Variante funktioniert das wie folgt: Das Programm im Userspace löst ein "supervisor call" (das ist ein Softwareinterrupt) aus, der die im Kernelmodus laufende Interrupt Service Routine zum Agieren zwingt. Diese erkennt dann den erwünschten Syscall und liefert die dafür notwendigen Parameter an den Compiler. Sobald der seine Arbeit getätigt hat, wird durch die ISR wieder per return an die ursprüngliche Adresse im Programm und damit in den Usermode gesprungen.
3. Im Microkernel gibt es die Variante, dass per CALL Befehl aus dem Userspace in den Kernelspace gesprungen wird. Und dann sorgt der Kernel dazu, dass von hier aus in den richtigen Adressbereich des gewünschten Moduls gesprungen wird, das auch im Userspace läuft. Hat das aufgerufene Modul seine Arbeit getan, wird per RETURN von hier noch einmal in den Kernelspace gesprungen, um von dort dann auch per RETURN in den Userspace zum Ausgangsprogramm zurückzukehren.
4. In Minix setzt man auf Arbeitsteilung. Hier wird eine Anfrage an den Microkernel gesendet (SEND), und falls der Kernel die Aufgabe nicht selber erledigen kann, von dort weitergeleitet. Per RECIEVE wird dem dazu auserkorenen Prozess die Anfrage eingereicht, welcher im Userspace läuft. Hat dieser seine Aufgabe erledigt, so übermittelt er per SEND das wieder zurück an den Kernel. Von dort werden die Ergebnisse per RECIEVE von Ausgangsprozess eingefordert und übermittelt.

6. C Syntax

```
#include <stdlib.h> // Importiert stdlib.h
#include <string.h> // Importiert string.h
#include <stdio.h> //Lädt die Bibliothek mit
    Standardfunktionalität für Input/Output
#include <time.h> // Lädt die time-Bibliothek, benötigt für die
    Erstellung von Zufallszahlen.

#define BUFFER_SIZE 100 // definiert das Makro BUFFER_SIZE mit
    dem Wert 100
```

```
int main(int argc, char *argv[]) // main-Funktion, die die
    Paramter argc (mit Int-Wert) und argv (Array von Chars)
    übergeben bekommt und einen int-Wert zurückliefert
{
    int i, guess, target, goes, max = 100; // Deklariert die
        Variablen als int. max wird auch gleich mit dem Wert 100
        initialisiert
    char buffer[BUFFER_SIZE]; // Deklariert ein Char-Array buffer
        der Länge von BUFFER_SIZE (also 100)

    srand((unsigned) time(NULL)); // Initialisiert den random
        number generator.
    printf("Welcome!\n"); // schreibt Welcome mit Zeilensprung
    for(i = 0; i < 8; i++) { //For-Schleife, die von i=0 bis i=7
        läuft und nach jedem Durchlauf i um eins erhöht.
        printf("="); //druckt = aus
    }
    printf("\n\n"); //druckt zwei Zeilensprünge

    do { // leitet do-Schleife ein
        goes = 0; //setzt goes = 0
        target = rand() % max + 1; // berechnet target per
            Zufallszahl (rand() gibt zufällige Zahl des int-Bereichs
            aus). Durch modulo max + 1 nie größer als 100
        guess = 0; //setzt guess = 0
        while(guess != target) { // solange guess ungleich target
            printf("Enter your guess: "); //druckt "Enter your guess"
            fgets(buffer, BUFFER_SIZE - 1, stdin); //Liest Eingabe aus
                stdin ein mit maximaler Länge von BUFFER_SIZE-1 pro Zeile
                und speichert das Ergebnis in buffer
            while(!scanf(buffer, "%d", &guess)) // while Schleife, die
                prüft ob in guess auch tatsächlich ein int-Wert
                eingegeben wurde
            {
                printf("Enter a number! "); // schreibt Enter a number in
                    die Konsole
                fgets(buffer, BUFFER_SIZE - 1, stdin); //Liest Eingabe aus
                    stdin ein, die die maximale Länge von BUFFER_SIZE-1 hat.
                    Die Eingabe wird dann nach buffer gespeichert
            }
            if(guess < target) // wenn der Wert in guess kleiner als der
                in target ...
                printf("Too low! "); // dann schreibe too low
```

```
    else if(guess > target) // andererseits, wenn guess größer
        als der Wert in target ...
        printf("Too high!\n"); // ... dann schreibe too high
    ++goes; // goes wird um eins inkrementiert (pre)
}
printf("Well done, it was %d!\n", target); // schreibt Well
done, it was <hier wird der Wert von target eingesetzt>!
mit Zeilenumbruch
printf("You took %d goes.\n\n", goes); // schreibt You took <
Wert von goes> goes. mit zwei Zeilenumbrüchen
printf("Another go? (y/n)\n"); // schreibt Another go? (y/n)
auf die Konsole.
fgets(buffer, BUFFER_SIZE - 1, stdin); // liest wieder eine
Zeile von stdin ein der Länge BUFFER_SIZE-1 und schreibts
nach buffer.
} while (buffer[0] == 'y' || buffer[0] == 'Y'); //Bedingung
für die do-Schleife. Sie läuft so lange, wie im Buffer im
Index 0 y oder Y steht.

printf("Goodbye!\n"); // Goodbye wird ausgegeben
return EXIT_SUCCESS; // gibt den Wert EXIT_SUCCESS zurück\left(
}
```

7. C Syntax

1. Gibt "Hello, World!" in einer for- Schleife aus.

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;
    for(i=0; i<5; i++)
        printf("Hello, World!\n");
    return EXIT_SUCCESS;
}
```

2. Gibt "Hello, World!" mit einer While-Schleife aus.

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
```

```
    int i = 0;
    while(i < 5)
    {
        printf("Hello , \uWorld!\n");
        i++;
    }
    return EXIT_SUCCESS;
}
```

3. Benutzt zur mehrfachen Ausgabe von "Hello, World!" eine do-while-Schleife.

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i = 0;
    do
    {
        printf("Hello , \uWorld!\n");
        i++;
    } while(i < 5);
    return EXIT_SUCCESS;
}
```

4. Gibt abwechselnd "Hello, World!" und "Goodbye, World!" aus mit if-else.

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;
    for(i=0; i < 10; i++)
    {
        if(i % 2 == 0)
            printf("Hello , \uWorld!\n");
        else
            printf("Goodbye , \uWorld!\n");
    }
    return EXIT_SUCCESS;
}
```

5. Gibt abwechselnd "Hello, World!" und "Goodbye, World!" aus durch Switch-Anweisung.

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;
    for(i=0; i<10; i++)
    {
        switch(i % 2)
        {
            case 0:
                printf("Hello , \uWorld!\n");
                break;
            case 1:
                printf("Goodbye , \uWorld!\n");
                break;
        }
    }
    return EXIT_SUCCESS;
}
```

6. Gibt zufällig "Hello, World!" und "Goodbye, World!" aus.

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int max = 2, i, target;
    srand((unsigned) time(NULL)); // Initialize random
    number generator.
    for(i=0; i<5; i++)
    {
        target = rand() % max + 1;
        if(target == 1)
            printf("Hello , \uWorld!\n");
        if(target == 2)
            printf("Goodbye , \uWorld!\n");
    }
    return EXIT_SUCCESS;
}
```

7. Gibt je nach Benutzereingabe "Hello" oder "Goodbye" aus.

```
#include <stdlib.h>
```



```
#include <stdio.h>

#define BUFFER_SIZE 100 // definiert die Buffer_size

int main(int argc, char *argv[])
{
    int i;
    char buffer[BUFFER_SIZE];
    for(i=0; i<5; i++)
    {
        printf("Hello?_(y/n)");
        fgets(buffer, BUFFER_SIZE - 1, stdin);
        if(buffer[0] == 'y' || buffer[0] == 'Y')
            printf("Hello\n");
        else
            printf("Goodbye\n");
    }
    return EXIT_SUCCESS;
}
```

8. Gibt so lange immer wieder "Hello" aus, bis "n" eingegeben wird.

```
#include <stdlib.h>
#include <stdio.h>

#define BUFFER_SIZE 100 // definiert die Buffer_size

int main(int argc, char *argv[])
{
    int i;
    char buffer[BUFFER_SIZE];
    do
    {
        printf("Hello ,_world?_(y/n)");
        fgets(buffer, BUFFER_SIZE - 1, stdin);
        printf("Hello!\n");
    }while(buffer[0] != 'n' && buffer[0] != 'N');
    printf("Goodbye ,_World!\n");
    return EXIT_SUCCESS;
}
```

8. Kommandozeilenparameter

Der Aufruf des Programms mit einer Reihe von Parametern führt dazu, dass die Parameter in umgekehrter Reihenfolge ausgegeben werden.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int i;
    for(i=argc-1; i>0; i--)
        printf("%s_", argv[i]);
    return EXIT_SUCCESS;
}
```