

## 1. Begriffe

**Prioritäteninversion** Bei der Prioritäteninversion wird ein Prozess höhere Priorität durch Systemumstände dazu gezwungen, auf einen Prozess von geringerer Priorität zu warten.

**Real-Time-Scheduling** Beim Real-Time-Scheduling hängt die Korrektheit des Systems nicht nur vom Ergebnis ab, sondern auch vom Zeitpunkt, wann das Ergebnis zu Verfügung gestellt wird. Daher werden an Real-Time-Scheduling-Algorithmen andere Anforderungen gestellt als normale Scheduling-Algorithmen.

**DMA** Bei dem Direct Memory Access wird die Kommunikation nicht über den Prozessor getätigt sondern über das DMA-Modul. So wird der Prozessor geschont und ist für andere Aufgaben frei; nach Beendung des Datentransfers sendet das Modul ein Interrupt an den Prozessor.

**Pipes** Eine Pipe ist eine Verbindung zwischen Prozessen, die es erlaubt, dass die Ausgabe eines Prozesses als Eingabe eines anderen verwendet wird.

## 2. Scheduling

### 3. Multiprozessor Scheduling

Die drei Varianten sind Global queue, Master / slave und Peer.

Bei Global queue gibt es wie der Name aussagt eine globale Warteschlange an die sich die Prozesse "anstellen" um von dort an einen beliebigen freien Prozessor zugeteilt zu werden. Bei dem zweiten Prinzip werden die Prozessoren und Prozesse in zwei Kategorien eingeteilt. So laufen die essentiellen Kernelfunktionen auf dem Masterprozessor, zusätzlich auch die Prozesse, die das Scheduling übernehmen. Für Master und Slave gibt es auch gesonderte Warteschlangen. Bei Peer gibt es nun für jeden Prozessor seine eigene Warteschlange. Jeder Prozessor muss sein Scheduling selber handhaben und alle Prozessoren sind gleichberechtigt, d.h. auf jedem beliebigen können die Kernel- und Betriebssystemkomponenten laufen.

### 4. I/O Abstraktionen

Als Beispiel für den blockbasierte Zugriff können Festplatten und Datenspeichermedien und Laufwerke allgemein genannt werden. Hingegen kommt der streamorientierte bzw. auch zeichenbasierte Zugriff für Peripheriegeräte wie Netzwerkadapter, Mouse oder auch ein Drucker in Frage.

An Hand der genannten Beispiele wird schon deutlich, dass bei den blockbasierten Zugriffen auf bereits in Mengen vorhandene Daten zugegriffen wird, so dass ein Einlesen von ganzen Blöcken fester Größe von Daten sinnvoll ist. Bei Mouse oder Netzwerkschnittstelle stehen andererseits die Daten nicht im Voraus zu Verfügung, sondern können nur fortlaufend am Stück in einem Datenstrom aufbereitet werden. Datenstrom sagt ja schon aus, dass es ein andauernder Zugriff ist, wohingegen beim blockbasierten immer nur ein Block gleichzeitig transportiert werden kann.

### 5. Fifty/Fifty

- *Die Laufzeit des Scheduling-Algorithmuses ist für die Echtzeitfähigkeit des Gesamtsystems von Bedeutung.*  
Ja. Bei Echtzeitsystemen ist es gerade wichtig, dass der Output zur richtigen Zeit vorliegt. Daher kann ein langsamer Scheduling-Algorithmus auslösend dafür sein, dass der Zeitplan nicht eingehalten werden kann.
- *In den Meta-Informationen zu einer Datei wird die Prozess ID des Prozesses, der zuletzt auf die Datei zugegriffen hat, hinterlegt.*  
Nein. Das würde bei der Prozess-ID auch keinen Sinn ergeben, da diese sich bei jedem Neustart des Prozesses ändert. Es würde mehr Sinn ergeben, den Prozessnamen oder das ausführende Programm zu speichern.

## 6. IO

Als erstes die Implementierung von cat

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

```
int main(int argc, char *argv[]) {  
    FILE* dat;  
    int c;  
    char *str;  
    if(fopen(argv[1], "r") != NULL) { // Kann Datei geöffnet  
        werden?  
        dat = fopen(argv[1], "r");  
        while((c = fgetc(dat)) != EOF) { // solange noch  
            Buchstaben vorhanden  
            sprintf(str, "%c", c); // in str c als char speichern  
            fputs(str, stdout); // Ausgabe auf stdout  
        }  
        fclose(dat); // Datei schließen  
    }  
    else { // Keine Datei, von stdin einlesen  
        while((c = fgetc(stdin)) != EOF) { // solange noch  
            Buchstaben auf der Konsole  
            sprintf(str, "%c", c);  
            fputs(str, stdout);  
        }  
    }  
    return 0;  
}
```

Und nun die von wc

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

```
int main(int argc, char *argv[]) {  
    FILE *dat;  
    int linecounter = 0;  
    int wordcounter = 0;  
    int bytecounter = 0;  
    int bool = 1;  
    int c;  
    if(fopen(argv[1], "r") != NULL) { // Kann Datei geöffnet  
        werden?  
        dat = fopen(argv[1], "r");  
        while((c = fgetc(dat)) != EOF) { // solange in der  
            Datei noch Zeichen sind
```

```
        bytecounter++; // für jedes Zeichen Byte zählen
        if(c == '\n') { // wenn Zeilenumbruch, dann
            Zeilenzähler erhöhen
            linecounter++;
        }
        if(c == ' ' || c == '\n') { // wenn Leerzeichen
            oder Zeilenende, merke dies in bool
            bool = 1;
        }
        else { // anderes Zeichen
            if (bool == 1) { // wenn bool 1 ist, dann war
                vorher ein Leerzeichen, also neues Wort
                zählen, dabei bool zurücksetzen
                wordcounter++;
                bool = 0;
            }
        }
    }
    fclose(dat); // Datei schließen
}
else { // Datei konnte nicht geöffnet werden, alles jetzt
    von stdin
    while((c = fgetc(stdin)) != EOF) {
        bytecounter++;
        if(c == '\n')
            linecounter++;
        if(c == ' ' || c == '\n')
            bool = 1;
        else {
            if(bool == 1) {
                wordcounter++;
                bool = 0;
            }
        }
    }
}
// Ausgabe
char *str;
sprintf(str, "%d lines and %d words and %d byte\n",
        linecounter, wordcounter, bytecounter);
fputs((" %s", str), stdout);
return 0;
}
```

Und zu guter letzt von grep

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

#define BUFF 10000

int main(int argc, char *argv[]) {
    FILE *dat;
    char *line;
    char *str;
    line = malloc(BUFF);
    if(fopen(argv[2], "r") != NULL && argc >= 3) { // Kann
        Datei geöffnet werden?
        dat = fopen(argv[2], "r");
        str = argv[1]; // zu suchender String
        while((fgets(line, BUFF, dat)) != NULL) { // solange es
            noch Zeilen mit Maximaler Länge von BUFF gibt,
            speichere diese in line ...
            if(strstr(line, str) != NULL) // wenn str in line
                vorkommt
                fputs("%s", line), stdout); //dann gebe line auf
                stdout aus
        }
        fclose(dat); // Datei wieder schließen
    }
    else if(argc == 2) { // wenn Datei nicht geöffnet werden
        kann, lese von stdin ein, der erste Parameter ist das
        Suchwort
        str = argv[1];
        while((fgets(line, BUFF, stdin)) != NULL) {
            if(strstr(line, str) != NULL)
                fputs("%s", line), stdout);
        }
    }
    else // Kein String zum Suchen eingegeben -> Fehler
        fputs("Fehler, kein Suchstring eingegeben\n", stderr);
    return 0;
}
```

## 7. Speicherverwaltung

Um die exakte Funktionsweise der Operatoren zu erklären, wird am besten erstmal der Quelltext an entsprechenden Stellen kommentiert:

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

#define SIZE 10
#define ASCII_OFFSET 65

int main(int argc, char *argv[]) {
    char* ptr; //Ein Pointer namens ptr auf ein char-Objekt
    char* tmp; //Ein Pointer mit Namen tmp auf ein char-Objekt
    char array[SIZE]; // Deklariert ein Feld von char mit der
        Größe von Size
    int i;

    for(i = 0; i < SIZE; i++)
        array[i] = i + ASCII_OFFSET; //speichert in die Speicherzelle
            i von dem reservierten Feld den Wert i+ASCII_OFFSET (das
            sind hier die 9 Großbuchstaben von A bis I)
    array[SIZE - 1] = '\0';
    printf("array is \"%s\".\n", array); // mit array alleine wird
        der Pointer auf das erste Element des Felder bezeichnet,
        es wird also das ganze Feld als String ausgegeben

    ptr = array; // der Pointer ptr erhält den Pointer array
    printf("ptr points to \"%s\".\n", ptr); //da ptr auf array
        zeigt, wird hier wieder das ganze Feld als String
        ausgegeben

    printf("(array+5) is \"%s\".\n", array + 5); // addiert zu
        der Adresse des ersten Elements vom Feld +5, und landet
        somit beim 6. Element (F). Ab hier wird dann der ganze
        String ausgegeben
    printf("&array[5] is \"%s\".\n", &array[5]); //Mit & wird die
        Adresse des Objektes zurückgeliefert, also hier die Adresse
        vom Element mit Index 5 (das ist das 6. Element), also F

    ptr = (char*) malloc(SIZE * sizeof(char)); //reserviert einen
        Speicherbereich der Größe SIZE für char-Elemente und
        übergibt dann den Zeiger darauf an ptr
    if(ptr == NULL) {
```

```
    printf("Can't allocate enough memory.\n");  
    return EXIT_FAILURE;  
}  
i = 0;  
tmp = ptr; // der Zeiger von tmp zeigt auf den selben Bereich  
            wie der von ptr  
while(array[i]) // greife auf char in Speicherzelle i zu  
    *(tmp++) = array[i++]; // das * folgt dem Zeiger auf das  
                           tatsächliche char-Objekt und weißt ihm den Wert von array[  
                           i++] zu, dabei wird mit tmp++ die Adresse des momentanen  
                           tmp auch immer erhöht  
printf("ptr points to \"%s\".\n", ptr); // gibt wieder das  
    ganze Feld aus, da ptr und tmp auf das gleiche Objekt  
    zeigen  
free(ptr);  
  
return EXIT_SUCCESS;  
}
```

Um es nochmal kurz zusammenzufassen, der \*-Operator versteht sich als Pointer auf ein Objekt, das heißt in ihm ist nur eine Speicheradresse hinterlegt. Mit dem &-Operator kann man nun von einem Objekt die Adresse erhalten, an der es sich befindet. Die Klammern eines Arrays sind nichts anderes als eine andere Schreibweise für Pointer auf hintereinanderliegende Speicherbereiche gleichen Typs. Man spart sich hier immer über die Adresse zu gehen, sonder kann so direkt auf das jeweilige Element zugreifen.