4 Gerätetreiber

Gerätetreiber *(device driver)*

- ermöglicht höheren BS-Schichten die komfortable Nutzung von E/A-Geräten einer bestimmten Klasse,
- bietet zu diesem Zweck eine geeignete Schnittstelle zur Erteilung von E/A-Aufträgen,
- besorgt die sekundäre Unterbrechungsbehandlung der E/A-Geräte.

Terminologie zur Hardware:

Gerätesteuerung *(device controller):*

Teil der Peripherie, zuständig für Ansteuerung einer *Gruppe gleichartiger* Ein/Ausgabegeräte

Ein/Ausgabeoperation:

Ein/Ausgabe eines Zeichens, einer Zeile, eines Blocks usw. (je nach Geräteart)

Geräteregister (device registers):

Steuerregister zum Konfigurieren, Starten etc.,
Statusregister zum Abfragen von Fehlern etc.

4.1 Aufgaben eines Treibers

❶ **Auftragspufferung**

❷ **Auftragssteuerung** *(device scheduling)*

❸ **Fehlerbehandlung**

❹ **Benachrichtigung des Auftragggebers**

❶ Auftragspufferung:

Es macht Sinn, für die eintreffenden E/A-Aufträge beim Treiber eine *Warteschlange* zu führen: er kann damit bei Beendigung einer Ein/Ausgabeoperation als Teil der Unterbrechungsbehandlung sehr schnell die nächsten Operation starten – was für eine gute Ausnutzung des Gerätes sorgt.

Bemerkung: Leistungsfähige Gerätesteuerungen übernehmen diese Auftragspufferung ganz oder teilweise selbst.

❷ **Auftragssteuerung** *(device scheduling)*

Die Warteschlange muss nicht unbedingt FCFS abgearbeitet werden! Es kann geschickter sein, Aufträge in einer anderen Reihenfolge als der ihres Eintreffens an das Gerät weiterzugeben.

(Beispiel: wenn für eine Spur – bzw. einen Zylinder – eines rotierenden Speichers mehrere Aufträge für das Lesen/Schreiben von Blöcken/Sektoren vorliegen, sollten sie in der Reihenfolge ihres Eintreffens *unter dem Lese/Schreibkopf* bearbeitet werden.)

Bemerkung: Auch das wird häufig von Gerätesteuerungen übernommen.

❸ Fehlerbehandlung

Nach Erledigung eines E/A-Operation wird typischerweise das Statusregister befragt, ob die Operation erfolgreich ausgeführt wurde und – wenn nicht – welche Art Fehler vorliegt.

Es kann dann eine geeignete Fehlerbeschreibung an den Auftraggeber zurückgeliefert werden.

Bei manchen Geräten macht es Sinn, den E/A-Versuch zu wiederholen (z.B. bei Lesefehler auf Diskette)
- eine typische Aufgabe des Treibers.

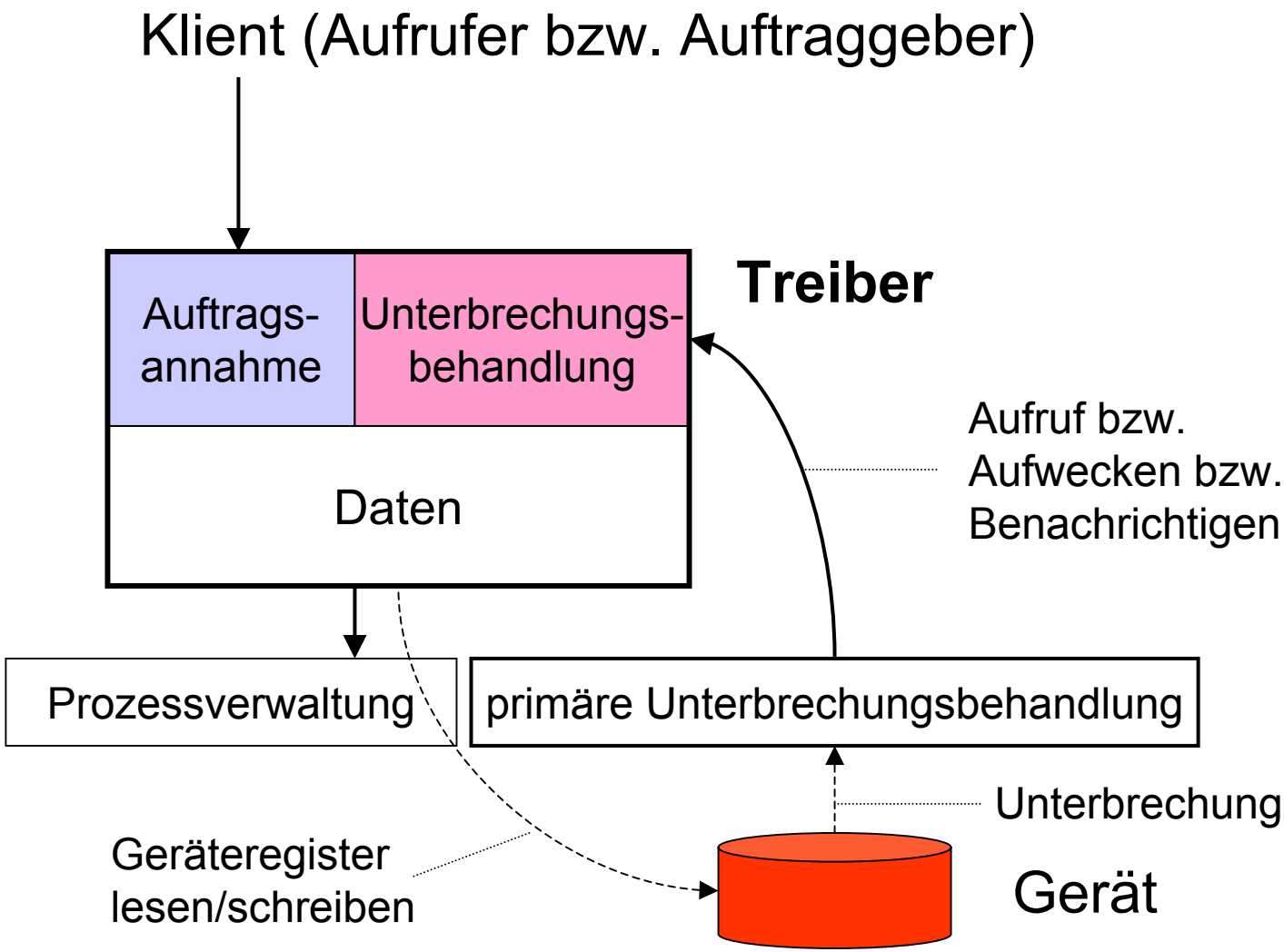
❹ Benachrichtigung des Auftraggebers

In den meisten Fällen ist der Auftraggeber an einer Bestätigung der erfolgreichen E/A interessiert bzw. möchte im Fehlerfall entsprechend informiert werden. Er wartet auf die Erledigung des Auftrags.

→ Aufwecken des Auftraggebers und Übermittlung von Ergebnis-Information erforderlich.

Bemerkung: Das schließt nicht aus, dass die Aufträge letztlich einer Quelle entstammen (z.B. einem Benutzerprozess), die eine asynchrone E/A-Schnittstelle benutzt (z.B. zu Dateisystem mit Ausgabe-Pufferung).

4.2 Einfaches Treiber-Modell



Einfachstmöglicher Treiber:

- nur für ein Gerät zuständig
- keine Auftragspufferung
- keine Auftragssteuerung
- keine Fehlerbehandlung

„Pseudocode“ für Gerät und Treiber:

```
interface DEVICE { void   start(Param p);
                   Status status();
}
```

```
class DeviceDriver {

      DEVICE device;      // controller hardware:
                          // device registers
      Status status;      // IO result
      Process client;     // current IO client
      Sema mutex = new Sema(1);
                          // one client at a time

public DeviceDriver(...) {
       ..... // initialize controller
}
```

```
public Result IO(Param p){ // client interface
       mutex.P();
       client = Process.current;

       DISABLE_INTERRUPTS();
       device.start(p);  // incl. assembly code
       client.block();
       ENABLE_INTERRUPTS();

       Status s = status;
       mutex.V();
       return s;
}
```

Beachte: Unterbrechungsunterdrückung garantiert, dass

1. `block` unteilbar ausgeführt wird (3.2.2🡐),
2. zwischen **`start`** und `block` kein Prozesswechsel erfolgt – denn dann könnte das Aufwecken (s.u.) vor dem Blockieren eintreten und damit wirkungslos bleiben!

Beachte ferner:

Die Parameterbehandlung und die Fehlerbehandlung ist i.a. erheblich komplizierter als es hier aussieht. (Übermittlung der Adressen von Pufferbereichen für die Ein/Ausgabe, Kopieren zwischen Adressräumen etc.)

```
public void IOcomplete(){ // interrupt handler
        // entered with interrupts disabled!

        status = device.status();
        client.wake();
}
} // end of class DeviceDriver
```

Beachte: Diese Operation `IOcomplete` ist lediglich der Kernbestandteil der sekundären Unterbrechungsbehandlung. Wir abstrahieren hier von der Art der Aktivierung (3.5←)

4.3 Auftragspufferung

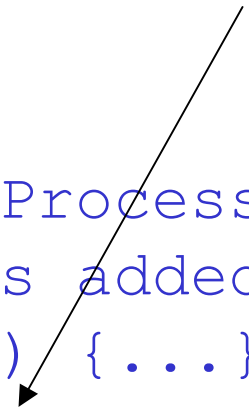
- Berücksichtigung der Geräte/Prozessorpriorität:

`GET_PRIORITY();`	liest Prozessorpriorität
`SET_PRIORITY(priority);`	setzt Prozessorpriorität

- Auftragsliste ist nicht unbedingt Warte*schlange :*

Auftragsliste:

(Dringlichkeit gemäß implementierter Auftragssteuerung)



```
class Commands {
      .....
public add(Param par, Process proc) {...}
       // (par,proc) is added as new entry
public Process remove() {...}
       // removes most urgent entry
       // and delivers its proc component (!)
public Process second() {...}
       // delivers par component (!)
       // of second most urgent entry
public void putfirst(Status s) {...}
       // places s in par component (!)
       // of most urgent entry
} // end of class Commands
```

```
class Driver {
      .....
      DEVICE device;
      int priority;  // device priority

      Commands commmands = new Commands();

.....
```

```
public void IO(Param par) {
        Process me = Process.current;
        int processorPrio = GET_PRIORITY();

        SET_PRIORITY(priority);
            commands.add(par,me);
            if(commands.length() == 1)
               device.start(par);

            DISABLE_INTERRUPTS();
               me.block();
            ENABLE_INTERRUPTS();

        SET_PRIORITY(processorPrio);
}
```

```
public void IOcomplete(){ // interrupt handler
        // entered with
        // processor priority = priority !

        Status s = device.status();
  // ! start new operation as soon as possible:
        if(commands.length() > 1)
            device.start(commands.second());
        commands.putfirst(s);
        Process client = commands.remove();
        DISABLE_INTERRUPTS();          (3.5.3←)
            client.wake();
        ENABLE_INTERRUPTS();
}
} // end of class Driver
```

4.4 Fehlerbehandlung

durch wiederholte Lese/Schreibversuche (wo sinnvoll)

Erweiterter Treiber:

```
class Driver {
      .....
      int maxretries = 3;
      int retries = 0;

public void IO(Param par) {.....}
```

```
public void IOcomplete(){ // interrupt handler
        // entered with
        // processor priority = priority !

        Status s = device.status();
        if(error(s) && retries < maxretries){
             device.start(commands.first());
             retries++; }
        else{retries = 0;
             if(commands.length() > 1)
                device.start(commands.second());
             commands.putfirst(s);
             ......}   (siehe 4.3)
}
} // end of class Driver
```

4.5 Strategien zur Auftragssteuerung

sind interessant bei rotierenden Speichern:

Latenzzeit *(latency time)* =

Zeit, die zwischen Start der E/A-Operation und dem tatsächlichen Beginn der Datenübertragung verstreicht

Übertragungszeit *(transfer time)* =

Zeit vom Beginn bis zum Ende der Datenübertragung

Ausführungszeit einer E/A-Operation

Rotation:

Rotationslatenz (rotational latency) (max. 10-200 ms)
+ Übertragungszeit

Armbewegung:

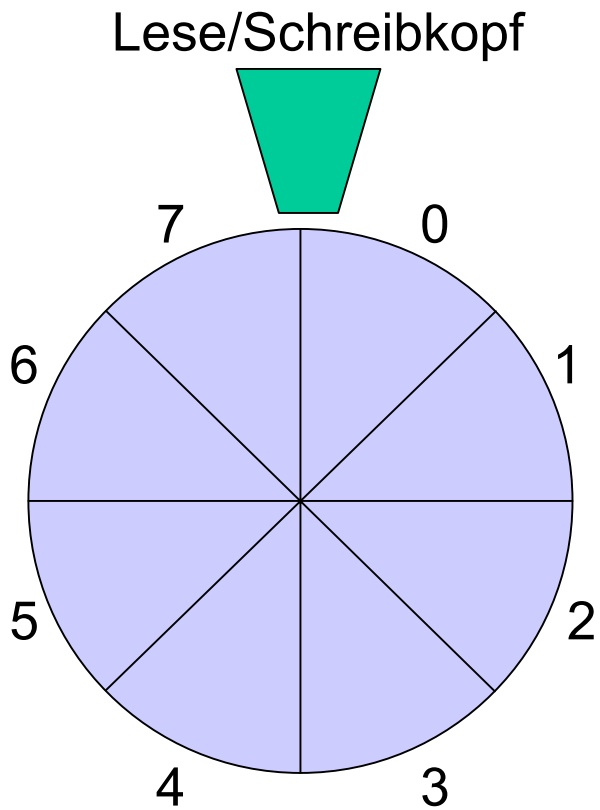
Positionierungslatenz (seek time) (max. 20-200 ms)
+ Rotationslatenz
+ Übertragungszeit

4.5.1 Berücksichtigung der Rotation

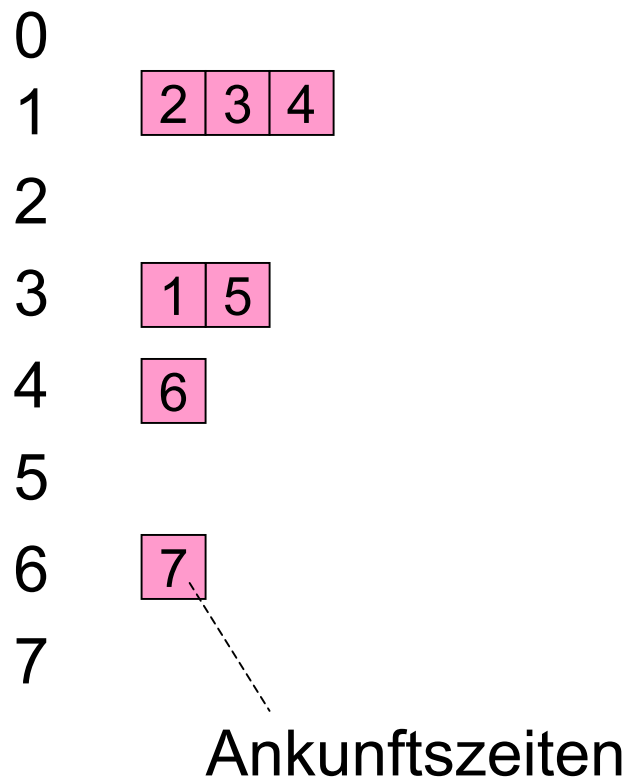
SLF *(shortest latency first)*

bedient die anstehenden Aufträge in derjenigen Reihenfolge, in der die entsprechenden Sektoren/Blöcke am Lese/Schreibkopf vorbeikommen – bei gleichem Sektor entscheidet die Wartezeit.

Beachte: Diese Strategie ist zwar nicht FCFS, aber trotzdem „hinreichend“ *fair:* ein Auftrag für den Sektor s wird spätestens nach so vielen Umdrehungen bedient wie ältere Aufträge für s vorliegen.



Auftragsliste ist organisiert als Folge von **Sektorschlangen**, die reihum bedient werden:



4.5.2 Berücksichtigung der Armbewegung

(disk head scheduling)

Verschiedene Strategien möglich:

SSTF *(shortest seek time first):*

bedient wird derjenige Auftrag, dessen Zylinder/Spur der augenblicklichen Armposition am nächsten liegt

→ effizient, aber *nicht fair!*

SCAN, auch *Fahrstuhl-Algorithmus (elevator algorithm):*

der Arm wird abwechselnd hin- und herbewegt,
und dabei werden die auf dem Weg liegenden
Aufträge erledigt.

→ effizient und fair – aber nicht gleichmäßig fair:
relativ *große Streuung der Wartezeiten*
(vgl. „Schneeschieber“)

CSCAN *(circular SCAN)* oder **C-LOOK**:

„der Fahrstuhl läuft grundsätzlich nur aufwärts“,
entspricht einem „aufgeschnittenen SLF“ (4.5.1)

→ gute Eigenschaften!

Pragmatische Aspekte:

❶ Kombination von SLF und CSCAN kaum sinnvoll:

gut ausgelegtes System hat kurze Auftragslisten

→ mehr als ein Auftrag für einen Zylinder
sehr unwahrscheinlich → CSCAN genügt.

❷ MINIX praktiziert sogar nur FIFO;

allerdings kann „Auftragspaket“ mehrere Sektoren betreffen – die dann in der passenden Reihenfolge bedient werden.

❸ Leistungsfähige Gerätesteuerungen machen alles selbst!