Chapter 8

I/O Devices
8.1 Input/Output Basics

**Processor (programmed I/O)**
The processor reads or writes data in bytes or words from/into a register in the controller.

**Direct Memory Access (DMA)**
The controller can autonomously access the memory via the bus.
Control

• **Triggering** (How does the controller get the requests?)

• The processor loads the corresponding register in the particular controller:
  • Type of operation (e.g. read, write)
  • Source
  • Target
  • Status

• **Reaction**
  • After completion of the I/O operation, the processor needs to be informed.
  • Two possibilities:
    • The processor checks occasionally the controller’s status register (**Polling**). (In most cases too inefficient.)
    • The processor gets informed about the completion by a special signal (**Interrupt**). (Usual approach)
Interrupts

- There is at least one interrupt wire for the processor.
- After each instruction the processor checks whether there is a signal (corresponding voltage level) at this wire.
- If so, it immediately (if interrupts are enabled) jumps to a subroutine that evaluates the interrupt and performs or triggers the necessary actions.
Interrupt analysis

• First, we only know the very fact of an interrupt.

• Therefore, we need to find out,
  - **who** (which device) caused the interrupt (source),
  - **why** the interrupt has happened (e.g. end of transmission, error).

• Thus, a subroutine for interrupt handling has the following structure:
Interrupt processing

- An interrupt can occur at any time and in any situation.
- Especially also during an interrupt processing!
- Two approaches:
  - Sequential interrupt processing (FCFS)
  - Nested interrupt processing
- The interrupt mechanism is also used for processor internal (synchronous) events, at which we need immediate reaction (division by zero, address violation). In that case, it is usually called **exception**.
Sequential interrupt processing

- Prevent further interrupts during interrupt processing (disable interrupt).
- The prevention can be limited to specific types of interrupts (masking).

The new interrupt B is processed only when the processing of the current interrupt A has been completed.
Nested interrupt processing

- Interrupts are classified into different priority classes according to their type.
- Interrupts of higher priority may interrupt the processing of lower priority interrupts.
Nested Interrupts

Ring Ring Ring

Hello, Sam. Would I like to go out with you Saturday?

Hold on, Sam. I’ll be right back.

Telegram for Judy Smith. Thankyou.

Thanks for waiting, Sam. Sure I’d love to go out with you. See you at 8 p.m.

Bye.
Actions during interrupt processing

The first part corresponds to a regular subroutine call:

1. save return address (next instruction, PC+1) to stack
2. load program counter (PC) with entry address of interrupt handling routine (from interrupt vector table that contains the addresses of all interrupt handlers)

Under control of interrupt handler:

3. save all register contents to stack
4. perform necessary actions according to type of interrupt
5. load register contents from stack
6. return to interrupted program (by loading the return address into program counter)
8.2 Tasks of I/O

- We deal with that part of the OS that is responsible for the operation of I/O-devices, e.g.
  - Keyboard, monitor, mouse
  - Hard disks
  - Scanner, Printer
  - Measuring probes, A/D-transformer
  - Network
  - ...

- To operate an device
  - Control positioning of movable parts, set device parameters, read status
  - Transport copy data from central unit (processor, memory) to device (or vice versa)
### Diversity of devices

<table>
<thead>
<tr>
<th>Device</th>
<th>Purpose</th>
<th>Partner</th>
<th>Data Rate (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyboard</td>
<td>input</td>
<td>human</td>
<td>0.00001</td>
</tr>
<tr>
<td>Mouse</td>
<td>input</td>
<td>human</td>
<td>0.00004</td>
</tr>
<tr>
<td>Laser printer</td>
<td>output</td>
<td>human</td>
<td>0.3</td>
</tr>
<tr>
<td>Voice output</td>
<td>output</td>
<td>human</td>
<td>0.6</td>
</tr>
<tr>
<td>Network-LAN</td>
<td>in-/output</td>
<td>machine</td>
<td>10-10000</td>
</tr>
<tr>
<td>Mass storage</td>
<td>storage</td>
<td>machine</td>
<td>5-600</td>
</tr>
<tr>
<td>Graphic display</td>
<td>output</td>
<td>human</td>
<td>100-16000</td>
</tr>
</tbody>
</table>

Data rates span several orders of magnitude!
**Control**

- Examples
  - set baud rate
  - position read/write head
  - perform page feed
  - rewind tape
  - position tape to EOT label
  - extract CD-sledge
  - ....
Data transport request

- We distinguish between
  - the transport request (trigger)
  - the real data transport
- Depending on the type of device more or less bytes are transmitted.
  - If it is only a few bytes, they may be included in the request itself:
    - "output character X ".
    - Transport request and data transport are the same.
  - If it is many bytes, the request is of the form:
    - "output those bytes that can be found in memory from address d_start to d_end."
  - In this case, the transport request is separated from the actual data transport.
Portioning

• Data transport usually takes place in smaller portions

• Examples
  • Bit
    single wire
  • Byte (character)
    byte serial interface
  • Word
    word parallel interface
  • Record (several bytes)
    devices with special geometry (line printer, page printer)
  • Block (typical: 512 Byte - 8Kbyte)
    between storage devices
Handling a transport request

The relation between the control unit (CU) and the device is obviously a **service** relation.
Transport request: Parameter

- Direction (read or write)?
- Where are the data (source s)?
- Where to move the data (target t)?
- How many data (length L)?
Integration of device activity into SW service structure

- Problem: How can a device send a message to a thread in order to deblock it?

Here should the thread be blocked

Here should the thread be deblocked
Modelling device activity

- as procedure (short activities only)
  
  I/O instruction - call(...) 
  
  The device can be regarded as a processor that executes exactly one thread. No switching. Active waiting for requests.

- as thread (for longer activities)
  
  Send request 
  
  The device can be regarded as a processor that executes exactly one thread. No switching. Active waiting for requests.
Integration of device activity into thread interaction

- **Solution:**
  - Device sends interrupt upon completion.
  - Processor, busy with any activity, gets interrupted and calls the respective interrupt processing routine (IPR).
  - IPR generates respective message and sends it to the requesting thread (to the channel, at which the thread is waiting).
Response

Each request processing must be checked. Many things can happen. The device provides information. In case of errors we need dedicated error processing.

Examples:

- Wrong transport parameter: memory address, track number, head number, block number
- Wrong operational states: device unknown, no voltage, no storage media, mechanical jam, no formatting
- Transmission error: parity error, synchronization error
- Media error: destroyed magnetic surface, head crash
Technical integration

Control and transport parameter as well as indications and notifications need to flow across device boundaries: There are basically two possibilities:

- **Deposit in memory**
  - Dedicated memory addresses are used as registers for device communication (*memory-mapped I/O*). They can be read and written by the device.

- **Deposit in device**
  - The device disposes of registers that can be read and written by the processor using special I/O instructions.

- Independent of the realization we can model these I/O registers e.g. in the following way:

```c
struct IO_register {
    PARAMETER_OF_REQUEST IOR;
    START_SIGNAL IOS;
    ERROR_CODE IOE;
    FINISHED_INDICATION IOF;
};
```
8.4 Driver

- To relieve the programmer of tedious details, all device specific activities should be confined to one single component.
- This component is called **driver**.
Realization of a driver

**a) as procedure**

```
application program
  call
```

```
driver
  S
  R
```

```
device
  R
```

**b) as thread**

```
application program
  S
  R
```

```
driver
  S
  R
```

```
device
  R
```

Barry Linnert, linnert@inf.fu-berlin.de, Betriebssysteme WS 2019/20
Interaction between driver and user thread

a) Parallelism between user thread and driver

b) Buffering
Buffering

Source: Stallings, chap. 11

Barry Linnert, linnert@inf.fu-berlin.de, Betriebssysteme WS 2019/20
Coarse structure of a driver thread

Parameter preprocessing:
- Grab parameters
- Decompose, if necessary
- Load into I/O registers

Progress analysis

From user thread to device from device

To user thread
Error handling

• The progress analysis mainly checks, if there were errors. If yes, action is taken, if possible.
• Regarding error handling we can distinguish three types of errors:
  • Delaying:
    Can be fixed by support of user:
    Examples:  *paper tray empty, no media in drive*
  • Stochastic:
    Randomly occurring failures can be cured by repetition.
    Examples:  *parity error, time-out, collision*
  • Final:
    If the error is not fixable, the request must be aborted:
    Examples:  *unknown device address, no voltage*
Aggregation / Disaggregation

- For devices operating byte-wise the driver often performs aggregation or disaggregation, resp., i.e. a larger set of individual characters are aggregated to a block or a block is disaggregated into individual characters.

- Additionally some other tasks are performed:
  - For output, control characters are inserted (e.g. end of record, line feed, block number).
  - For input, control characters are filtered out (e.g. delete character).
Aggregation

Aggregation at input

Disaggregation at output
8.5 Strategies for mass storage drivers

- A driver accepts requests from many different user threads.
- Thus, many requests may line up at its entry channel.

- Usually, requests are processed in the order of their arrival (FCFS, FIFO).
- **Exception**: mass storage driver
8.5.1 Hard Disk Drives (HDDs)

- A hard disk consists of one or more platters with magnetizable surfaces that rotate at high speed (ca. 3000-10000 rpm).
- For each magnetizable surface there is a **read-write head** that can be moved back and forth across the whole surface. The gadget that holds the read/write heads is called disk **arm**.
- If the disk device consists of more than one platter, the different arms are attached to a **broom**.
- Each surface consists of concentric **tracks** (1000-20000), on which the information is recorded.
- Each track in turn is composed of many **sector** (50-800).
- Each sector can store a specific amount of data (normally 0.5 or 4 kByte).
- The entirety of all tracks that can be read/written with the same arm position, is called **cylinder**. (In case of only one surface, cylinder and track are the same.)
Geometry of a hard disk

cylinder
read/write head
arm
broom
arm/broom movement

cylinder

Sectors
Example: Hitachi Travelstar 7K500

- Capacity (formatted): 320 GB
- Bytes per Sector: 512 Bytes
- Positioning time (next track): 1 Milliseconds
- Positioning time (average): 12 Milliseconds
- Positioning time (max): 20 Milliseconds
- Average latency: 4.2 Milliseconds
- Rotational speed: 7,200 rpm
- Transmission rate (max.): 1245 Mbit per sec
- Internal buffer: 16 MB
- Storage density (max.): 370 Gbit/sq.in.
Parameters

- The service time $t_{\text{serv}}$ of a disk request \((op, n, s)\) to read or write from sector \(s\) on track \(n\) is additively composed of the following components:
  
  **Arm positioning time** $t_{\text{pos}}$  
  time needed to move the arm from its current position to position \(n\)

  **Latency** $t_{\text{lat}}$  
  waiting time until the target sector appears under the head (rotational waiting time)

  **Read/write time** $t_{\text{read}}$  
  time for reading /writing the target sector

  **Transfer time** $t_{\text{trans}}$  
  time to transmit the block from or to main memory

- In addition to this service time $t_{\text{serv}}$ we have to calculate the waiting time $t_{\text{wait}}$ to get the total response time $t_{\text{resp}}$:

![Diagram showing the components of the total response time](image)
Example

• Assume we have a hard disk with the following parameters:
  • Arm positioning time 10 ms
  • Rotational speed 10,000 rpm
  • Sector size 512 bytes
  • Sector number 320/track

• Read request with following parameters:
  • 2560 Sectors, i.e. 1,3 Mbytes

• Question: What is the service time?
Example

- Case 1:
  File is contiguously stored, i.e. it allocates all sectors on 8 adjacent tracks (8 x 320 = 2560 sectors).

  Arm positioning time 10 ms
  Latency 3 ms  (half rotation)
  Reading 320 sectors 6 ms  (= 1/ (10.000 / 60) sec)

  Total 19 ms  for first track

  The remaining tracks can be read without additional delay, i.e. the average positioning time is neglectable (zero).
  Thus, for each remaining track we get 6+3=9 ms.

  In total we calculate a service time* of 19 + 7x9 = \textbf{82 ms}

* without transfer time to copy data into main memory
Example

- Case 2:
  File is stored randomly, i.e. it occupies sectors on any tracks.

  Arm positioning time       10 ms
  Latency                   3 ms
  Reading 1 sector 0.01875 ms (6 ms / 320)

  Total 13.01875 ms for each sector

Since we have to read 2560 sectors, we get a total service time of

  2560 x 13.01875 = 33,328 ms = **33.328 Seconds**

**Lesson learned:** Files should occupy contiguous sectors!
Strategies for hard disks

a) First come first served (FCFS)
   Processing requests in the order of their arrival

b) Shortest seek time first (SSTF)
   Always select request with the shortest arm positioning time

c) Elevator (SCAN)
   Like SSTF, but only in one direction

d) Cyclic Elevator strategy (SCAN-C)
   Like SCAN, but return to track 0 when reaching highest track number
Example FCFS

- Requests in queue (track number): 98, 183, 37, 122, 14, 124, 65, 67
- Current head position: 53
Example SSTF

- Requests in queue (track number): 98, 183, 37, 122, 14, 124, 65, 67
- Current head position: 53
Example SCAN (Elevator)

- Requests in queue (track number): 98, 183, 37, 122, 14, 124, 65, 67
- Current head position: 53
- Current direction: down
Example SCAN-C

- Requests in queue (track number): 98, 183, 37, 122, 14, 124, 65, 67
- Current head position: 53
### Example

#### (a) FIFO (starting at track 100)

<table>
<thead>
<tr>
<th>Next track accessed</th>
<th>Number of tracks traversed</th>
</tr>
</thead>
<tbody>
<tr>
<td>55</td>
<td>45</td>
</tr>
<tr>
<td>58</td>
<td>3</td>
</tr>
<tr>
<td>39</td>
<td>19</td>
</tr>
<tr>
<td>18</td>
<td>21</td>
</tr>
<tr>
<td>90</td>
<td>72</td>
</tr>
<tr>
<td>160</td>
<td>70</td>
</tr>
<tr>
<td>150</td>
<td>10</td>
</tr>
<tr>
<td>38</td>
<td>112</td>
</tr>
<tr>
<td>184</td>
<td>146</td>
</tr>
</tbody>
</table>

**Average seek length:** 55.3

#### (b) SSTF (starting at track 100)

<table>
<thead>
<tr>
<th>Next track accessed</th>
<th>Number of tracks traversed</th>
</tr>
</thead>
<tbody>
<tr>
<td>90</td>
<td>10</td>
</tr>
<tr>
<td>58</td>
<td>32</td>
</tr>
<tr>
<td>55</td>
<td>3</td>
</tr>
<tr>
<td>39</td>
<td>16</td>
</tr>
<tr>
<td>38</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>20</td>
</tr>
<tr>
<td>150</td>
<td>132</td>
</tr>
<tr>
<td>160</td>
<td>32</td>
</tr>
<tr>
<td>184</td>
<td>24</td>
</tr>
</tbody>
</table>

**Average seek length:** 27.5

#### (c) SCAN (starting at track 100, in the direction of increasing track number)

<table>
<thead>
<tr>
<th>Next track accessed</th>
<th>Number of tracks traversed</th>
</tr>
</thead>
<tbody>
<tr>
<td>150</td>
<td>50</td>
</tr>
<tr>
<td>160</td>
<td>10</td>
</tr>
<tr>
<td>184</td>
<td>24</td>
</tr>
<tr>
<td>90</td>
<td>94</td>
</tr>
<tr>
<td>58</td>
<td>32</td>
</tr>
<tr>
<td>55</td>
<td>3</td>
</tr>
<tr>
<td>39</td>
<td>16</td>
</tr>
<tr>
<td>38</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>20</td>
</tr>
</tbody>
</table>

**Average seek length:** 27.8

#### (d) C-SCAN (starting at track 100, in the direction of increasing track number)

<table>
<thead>
<tr>
<th>Next track accessed</th>
<th>Number of tracks traversed</th>
</tr>
</thead>
<tbody>
<tr>
<td>150</td>
<td>50</td>
</tr>
<tr>
<td>160</td>
<td>10</td>
</tr>
<tr>
<td>184</td>
<td>24</td>
</tr>
<tr>
<td>18</td>
<td>166</td>
</tr>
<tr>
<td>38</td>
<td>20</td>
</tr>
<tr>
<td>39</td>
<td>1</td>
</tr>
<tr>
<td>55</td>
<td>16</td>
</tr>
<tr>
<td>58</td>
<td>3</td>
</tr>
<tr>
<td>90</td>
<td>32</td>
</tr>
</tbody>
</table>

**Average seek length:** 35.8

---

**Source:** Stallings, chap. 11
Remarks

- FCFS requires on the average (uniform distribution) moving across 1/3 of the tracks.
- SCAN, SCAN-C, SSTF are advantageous for high load since (in the limit) we have to move only one track.
- SCAN and SSTF discriminate against marginal tracks (inmost and outmost).
- SSTF is generally the best. However, it can lead to starvation of request for marginal tracks.
Linux Deadline Scheduler

- The elevator strategy (SCAN) can lead to "starvation".
- Write accesses are usually asynchronous, read accesses synchronous. A stream of write accesses can therefore significantly delay a read access.
- Therefore, each request is furnished with a deadline and additionally (besides the SCAN-Queue) inserted into a read or write queue.
- In the normal case the SCAN queue is being processed, unless the first request of one of the FIFO queues becomes overdue (deadline expired). Then this request (and some more) of the respective FIFO queue are processed.
Realization of non-FCFS strategies

Sorting of requests according to track number can be done:

- In the communication operations at the driver’s entry channel. Means special variant of channel (receive).
- In the driver itself, i.e. all incoming request are admitted to the driver.

Requires parallelism between
- Request acceptance
- Request processing

_Pipeline!_
Driver as Pipeline

- From user:
  - Lock
  - Insert request according to track number
  - Unlock

- Next request in current direction
  - Lock
  - Unlock

- Small number of requests in data structure of driver
  - From device

- Large number of requests in data structure of driver
  - To device

- From user:
  - To device
  - From device
Delay time reduction

- According to the current state of discussion the driver gets blocked if it has submitted a request to the device (entering a synchronous receive).
- Deblocking takes place as part of the interrupt handling, when the request processing is finished.
- Even if the driver runs at high priority, some time elapses, until the driver can submit the next request.
Latency Hiding

- At sequential access to disk the time between two successive requests should not be longer than the time needed to move across a sector or block gap.
- Otherwise the next sector can be read only after one full rotation.

- We can reduce this latency if the driver prepares the next request during the time, when the device is processing the current request.

- To achieve this, the work has to be organized such that parallelism between device and driver is possible.

- The solution is the driver’s decomposition into two phases, where the first creates new requests and the second is responsible for progress analysis.
Two-Phase-Driver

- The access to device specific data (registers) is then a critical section that needs to be put under mutual exclusion (locking).
- It is also recommended to give the first driver phase higher priority (with preemption) to make sure that after releasing the lock the next request can be submitted immediately.
Read-ahead

- Read more than currently requested
  - In assumption the next sectors or blocks will be requested soon.
  - Reading of next sectors or blocks with small cost.

- Buffering of not yet requested blocks
  - Device cache
  - Buffer cache in main memory

- Not appropriate for writing!
  - Write back caches
8.5.2 Solid State Drives (SSDs)

- SSDs use integrated circuit assemblies as memory to store data persistently.
- SSDs do not employ moving mechanical components:
  - Shock and vibration resistant.
- Non-Volatile NAND-based flash memory commonly used, however SSDs can be constructed from RAM but need additional precautions against power loss.
- SSDs are available in the form factor of HDDs in both physical appearance and connectivity:
  - HDDs can be easily replaced by SSDs.
## HDDs vs. SSDs

<table>
<thead>
<tr>
<th>Property</th>
<th>HDD</th>
<th>SSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random access time</td>
<td>2.9ms to 12ms</td>
<td>&lt;0.1ms</td>
</tr>
<tr>
<td>Start-up time</td>
<td>Several seconds (disk spin-up)</td>
<td>Almost instantaneous</td>
</tr>
<tr>
<td>Read latency time</td>
<td>Different for every different seek</td>
<td>Generally low</td>
</tr>
<tr>
<td>Data transfer rate</td>
<td>~140 MB/s</td>
<td>100-600 MB/s (Highest-End several GB/s)</td>
</tr>
<tr>
<td>Read performance</td>
<td>Depends on required number of seeks</td>
<td>Consistent on whole SSD</td>
</tr>
<tr>
<td>Fragmentation</td>
<td>Problematic due to additional seeks</td>
<td>No Problem</td>
</tr>
<tr>
<td>Noise</td>
<td>Can be significant</td>
<td>(almost) none</td>
</tr>
<tr>
<td>Cost per capacity</td>
<td>&lt;$0.10 per GB</td>
<td>&gt;$0.50 per GB</td>
</tr>
<tr>
<td>Number of writes</td>
<td>~10^{10} per sector</td>
<td>~10^{4}-10^{6} per block</td>
</tr>
</tbody>
</table>
Physical Assembly of SSDs

- (logical block addresses) are mapped to flash pages by a controller

- Flash pages can be reassigned to different LBAs

- Only blocks of ~64-128 pages are erasable
No Updates In-Place

Read-/Write-Request

Flash

Temporary memory

empty page
valid page
invalid page
page write request
page delete request
Garbage Collection

write request

moire pages

occupied blocks

select block with lowest number of valid pages and clean it

empty block
Wear leveling

- Flash blocks wear out by writing to them
- Some flash blocks may reach their end of life much earlier than others
- Wear leveling distributes all write accesses over the medium

- Dynamic wear leveling
  - LBAs are initially mapped to a certain flash block
  - LBAs are mapped to a different flash block as soon as they are rewritten

- Static wear leveling
  - Additionally to dynamic wear leveling blocks are relocated and remapped when they are not written for a certain time
Write Amplification

• Write Amplification is a phenomenon where the actual amount of physical information written is a multiple of the logical information written

\[
\text{write amplification} = \frac{\text{data written to the (flash) memory}}{\text{data written by the host}}
\]

• Write Amplification gets increased by
  • Wear leveling
  • Random writes

• Write Amplification can be reduced by
  • Garbage collection
  • Over-provisioning
  • TRIM
Sustaining write performance

- Random write performance of an SSD is influenced by the workload history of the SSD.
- Main factor is the number of available empty pages in the SSD.

The number of available empty blocks can be increased by:
- Spare capacity
  - total capacity = spare capacity + usable capacity
  - A number of blocks that will never be used
  - Enforced by the SSD controller (spare capacity not visible for OS)
  - Enforced by the OS (spare capacity not visible for user)
- TRIM
  - TRIM is an ATA-Command that informs the device of invalid sectors
  - Allows an SSD to garbage collect blocks before their mapped sectors are rewritten
Flash in embedded/mobile devices

- Embedded CPUs are normally equip with or connected to flash memory similar to the flash memory in SSDs
  - Ranging from several kB e.g. in small 8-Bit controllers
  - To several GB e.g. in smart phones

- Flash memory is directly connected to the CPU – No controller!

- Most of the tasks of the SSD controller have to be done by the OS
  - Flash-Page mapping
  - Garbage collection
  - Wear leveling
8.6 RAID

- Access time for hard disks did not improve much compared to processor cycle times.
- The discrepancy of 6 orders of magnitude (msec vs. nsec) is still a significant performance obstacle.
- At the same time capacities have increased and costs plummeted.
- Therefore the question arises how we can get a performance benefit by using parallelism of inexpensive disks.
- RAID (Redundant Array of Independent Disks) is a standardized schema to organize file systems on several disks.
- Originally, RAID referred to „Redundant Array of Inexpensive Disks“. 
RAID

- Large numbers of disks mean an increased probability of failures.
- RAID has to be realized in fault-tolerant way.
- Properties
  - RAID is a set of physical hard disks, that appear to the user as one device.
  - The data is spread across the physical disks.
  - Redundant disk capacity may be used to store parity bits.
- RAID (originally) distinguishes 6 different schemes (RAID Level 0-5), later extended (RAID Level 6, combined levels such as 1-0 or 0-1)
- Fault model:
  If a block is requested, the disk delivers the correct block (fault free case) or an error message (in case of fault).
  RAID does not detect data corruption caused by disk (exception: RAID level 2 due to error correcting code) but relies on detection at level of disk (usually done by error correcting codes).
RAID Level 0 („non redundant“)

- Data organized in „Strips“ (e.g. sectors or blocks)
- Strips are distributed around on the disks.
- An I/O request consisting of different strips can be processed in parallel.
- No redundancy!

![Diagram of RAID Level 0]
RAID Level 1 („mirrored“)

- Fault tolerance by complete mirroring of all data.
- Costly, since double capacity required.
RAID Level 2 („Error Correcting Code“)

- Strips are small compared to RAID-0 or RAID-1.
- Typically bits or bytes.
- Error correcting codes (e.g. Hamming Codes) are calculated and stored on redundant disks.
- The number of redundant disks grows logarithmically with the number of disks.
- Disks need to be synchronized.
- Never used in commercial application.
RAID Level 3 („bit-interleaved parity“)

- Since we usually do know, which disk device has failed, we can use simple parity bits instead of Hamming-Codes.
- Only one redundant disk to store the parity bits is needed.
RAID Level 4
(„block-interleaved parity“)

- Like RAID Level 3, but with strips of block size
- Only one redundant disk to store parity bits required.
- Since each write operation needs to access the parity disk, there is a danger of a bottleneck.
- No disk synchronization needed.
RAID Level 5
(„block distributed parity“)

• Like RAID Level 4, but distribution of parity data across all disks
• Avoids bottleneck.
RAID Level 6 („dual redundancy“)

- RAID Level 5 can tolerate one failed disk at most.
- RAID Level 6 works with two independent checksum systems (P,Q) and can tolerate up to two disk failures.
- Parity information is stored in a distributed way like RAID Level 5.
## RAID Comparison

<table>
<thead>
<tr>
<th>Category</th>
<th>Level</th>
<th>Description</th>
<th>Disks Required</th>
<th>Data Availability</th>
<th>Large I/O Data Transfer Capacity</th>
<th>Small I/O Request Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Striping</td>
<td>0</td>
<td>Nonredundant</td>
<td>$N$</td>
<td>Lower than single disk</td>
<td>Very high</td>
<td>Very high for both read and write</td>
</tr>
<tr>
<td>Mirroring</td>
<td>1</td>
<td>Mirrored</td>
<td>$2N, 3N, \text{ etc.}$</td>
<td>Higher than RAID 2, 3, 4, or 5; lower than RAID 6</td>
<td>Highest of all listed alternatives</td>
<td>Up to twice that of a single disk for read; similar to single disk for write</td>
</tr>
<tr>
<td>Parallel access</td>
<td>2</td>
<td>Redundant via Hamming code</td>
<td>$N + m$</td>
<td>Much higher than single disk; higher than RAID 3, 4, or 5</td>
<td>Highest of all listed alternatives</td>
<td>Approximately twice that of a single disk</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Bit-interleaved parity</td>
<td>$N + 1$</td>
<td>Much higher than single disk; comparable to RAID 2, 4, or 5</td>
<td>Highest of all listed alternatives</td>
<td>Approximately twice that of a single disk</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Block-interleaved parity</td>
<td>$N + 1$</td>
<td>Much higher than single disk; comparable to RAID 2, 3, or 5</td>
<td>Similar to RAID 0 for read; significantly lower than single disk for write</td>
<td>Similar to RAID 0 for read; generally lower than single disk for write</td>
</tr>
<tr>
<td>Independent access</td>
<td>5</td>
<td>Block-interleaved distributed parity</td>
<td>$N + 1$</td>
<td>Much higher than single disk; comparable to RAID 2, 3, or 4</td>
<td>Similar to RAID 0 for read; lower than single disk for write</td>
<td>Similar to RAID 0 for read; significantly lower than single disk for write</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>Block-interleaved dual distributed parity</td>
<td>$N + 2$</td>
<td>Highest of all listed alternatives</td>
<td>Similar to RAID 0 for read; lower than RAID 5 for write</td>
<td>Similar to RAID 0 for read; significantly lower than RAID 5 for write</td>
</tr>
</tbody>
</table>

Source: Stallings, chap. 11
Further Reading

- Rubini, A.: Linux Device Drivers, 3rd ed. O'Reilly, 2005