



```
struct sched_param {
pthread_attr_t schedschedparam ( &a, &p ); // read parameter
// set p.sched_priority
pthread_attr_t schedschedparam ( &a, &p );
pthread_create ( &id, &a, thread_function, argument );
pthread_attr_destroy ( &a ); // destroy attributes
}
```

## 7. MUTEX UND SEMAPHORE

Jeder Thread hat seinen eigenen Instruction-Pointer und Stack-Pointer. Wenn Ergebnisse von der *Systemaufrufen* abhängigen Operationen abhängen, spricht man von einer *Race Condition* an. Threads müssen *synchronisiert* werden, damit keine *Race Condition* entsteht.

**Critical Section:** Code-Bereich, in dem Daten mit anderen Threads *geteilt* werden. Muss unbedingt synchronisiert werden.

**Atomare Instruktionen:** Eine atomare Instruktion kann vom Prozessor *unterbrechungsfrei* ausgeführt werden. *Achtung:* Selbst einzelne Assembly-Instruktionen *nicht* immer *atomar*

**Anforderungen an Synchronisations-Mechanismen: Gegenseitiger Ausschluss** (Nur ein Thread darf in Critical Section sein), **Fortschritt** (Entw. wenn in die Critical Section darf, muss in kritischer Zeit getroffen werden), **Begrenztes Warten** (Thread darf nicht über längere, bevor er in die Critical Section darf).

**Implementierung: Nur mit HW-Unterstützung möglich.** Es gibt zwei atomare Operationen, **Test-And-Sets** (Setzt einen int auf 1 und return den vorherigen Wert: `test_and_set(int * target) { int value = *target; *target = 1; return value; }`) und **Compare-and-Swap** (Überschreibt einen int mit einem spezifizierten Wert, wenn dieser dem erwarteten Wert entspricht: `compare_and_swap (int *a, int expected, int new_a) { int value = *a; if (value == expected) { *a = new_a; } return value; }`).

## 7.1. SEMAPHORE

Enthält Zähler  $z \geq 0$ . Wird nur über `Post(V)` (Erhöhe  $z$  um 1) und `Wait(V)` zugegriffen (Wenn  $z > 0$ , verringert  $z$  um 1 und fährt fort. Wenn  $z = 0$ , setzt den Thread in waiting, bis anderer Thread  $z$  erhöht).

`int sem_wait (sem_t *sem, int pshared, unsigned int value);` Initialisiert den Semaphore, typischerweise als globale Variable, `pshared = 1` Verwendung über mehrere Prozesse: `sem_t sem; int main (int argc, char ** argv) { sem_init (&sem, 0, 4); }` oder als Parameter für den Thread (Speicher auf dem Stack oder Heap): `STRUCT T { sem_t sem; ... ;`

`int sem_wait (sem_t *sem); int sem_post (sem_t *sem);` implementieren `Post` und `Wait`. `int sem_trywait (sem_t *sem); int sem_timedwait (sem_t *sem);` `sem_wait` `sem_post` `sem_timedwait` `abs_timeout`; sind wie `sem_wait`, aber `brechen` ab, falls Dekrement nicht durchgeführt werden sollte. `sem_wait` `sem_post` `sem_timedwait` sind wie `sem_wait`, aber `brechen` ab, falls Dekrement nicht durchgeführt werden sollte. `int sem_destroy (sem_t *sem);` Entfernt Speicher, den das OS mit `sem` assoziiert hat.

`semaphore free = 0;`  
`semaphore used = 0;`  
`while (1) { // Producer while (1) { // Consumer`  
`wait (free); // Hat es Platz in Queue? wait (used); // Hat es Elemente in Queue?`  
`produce_item (&buffer[i], ...); consume (&buffer[i], ...);`  
`POST (used); // 1 Element mehr in Queue POST (free); // 1 Element weniger in Queue`  
`m = (w+1) % BUFFER_SIZE; n = (r-1) % BUFFER_SIZE;`  
`}`

## 7.2. MUTEXE

Ein Mutex hat einen *binden* Zustand  $z$ , der nur durch zwei Funktionen verändert werden kann: *Acquire* (Wenn  $z = 0$ , setzt  $z$  auf 1 und fährt fort. Wenn  $z = 1$ , blockiert den Thread, bis  $z = 0$ ), *Release* (Setzt  $z = 0$ ). Auch als non-blocking-Function: `int pthread_mutex_trylock (pthread_mutex_t *mutex)`

`// Beispiel Initialisierung pthread_mutex_t mutex; // global void * thread_function (void * args) { int main() { // Enter critical section: pthread_mutex_lock (&mutex); // Leave critical section: pthread_mutex_unlock (&mutex); pthread_mutex_destroy (&mutex);`

**Priority Inversion:** Ein *hoch-priorisierter* Thread  $C$  wartet auf eine Ressource, die von einem *niedriger priorisierten* Thread  $A$  gehalten wird. Ein Thread mit Priorität zwischen diesen beiden Threads erhält den Prozessor. Kann mit **Priority Inheritance** gelöst werden. Die Priorität von  $A$  wird temporär auf die Priorität von  $C$  gesetzt, damit der Mutex schnell wieder freigegeben wird.

## 8. SIGNALS, PIPES UND SOCKETS

### 8.1. SIGNALS

Signale ermöglichen es, einen Prozess von *aussen* zu unterbrechen, wie ein *Interrupt*. *Unterbrechen* des gerade laufenden Prozesses/Threads, Auswahl und Ausführen der *Signal-Handler-Funktionen*, *Fortssetzen* des Prozesses. Werden über ungültige Instruktionen oder Abbruch auf Seitens Benutzer ausgelöst. Jeder Prozess hat pro Signal einen Handler.

**Alarm:** *Ignore-Handler* (ignoriert das Signal), *Terminate-Handler* (beendet das Programm), *Alarm-Termine-Handler* (beendet Programm, wenn Prozess in den Zustand *alarm* übergeht), *SIGSTOP* (Wie *SIGSTOP*, aber kann nicht ignoriert oder abgefangen werden), *SIGCONT* (Setzt den Prozess fort). **Signale von der Shell senden:** `kill 1234 5678` sendet SIGTERM an Prozesse 1234 und 5678

`int sigaction (int signal, struct sigaction *new, struct sigaction *old);` Definiert Signal-Handler für `signal`, wenn `new`  $\neq 0$ . (Die eigene Signal-Handler definiert via `sigaction` statt `sa_handler`: zu laufende Funktion, `sa_mask`: Blockierte Signale während Ausführung, `sa_handler` nur durch `sigset_t` (Funktion: `sigemptyset`, `sigfillset`, `sigdelset`, `sigisset`, `sigismember`)

### 8.2. PIPES

Eine geöffnete Datei entspricht einem *Eintrag* in der *File-Descriptor-Tabelle (FDT)* im Prozess. Zugriff über `File-AP` (bzw. `File-Descriptor`). Das OS speichert die *Eintrag der Prozess-FD* in einem *Verzeichnis auf die globale FDT*. Bei `fork()` wird die FDT auch kopiert.  
`int dup (int source_fd); int dup2 (int source_fd, int destination_fd);` Duplizieren den File-Descriptor `source_fd`. `dup2` allokiiert einen neuen FD, `dup2` überschreibt `destination_fd`.

**8.2.1. Umleiten des Ausgabestreams**  
`int fd = open ("log.txt", ...); int id = fork (); if (id == 0) { // child dup2 (fd, 1); // duplicate fd for log.txt as standard output // e.g., 'ls' will now write into 'log.txt' instead of stdout // else { // parent > close (fd); }`

Eine *Pipe* ist eine <Datei> (Eine Datei muss `open`, `close` etc. unterstützen) im Hauptspeicher, die über zwei File-Deskriptoren verbunden wird: *read* und *write* und *end*. Daten, die in *write* end geschrieben werden, können aus *read* end genau *einmal* und als *FIFO* gelesen werden. Pipes erlauben *Kommunikation über Prozess-Grenzen hinweg*. Ist unidirektional.

`int fd [2]; // 0 = read, 1 = write if (id == 0) { // Child close (fd [1]); // don't use write end close (buffer [BSIZE]); int n = read (fd[0], buffer, BSIZE); else { // parent > close (fd); } else { // don't use read end char * text = "A li lu"; write (fd [1], text, strlen(text) + 1); }`  
Pipe lebt so lange, wie mind. ein Ende geöffnet ist. Alle Read-Ends geschlossen  $\rightarrow$  SIGPIPE an Write-End. Mehrere Writes können zusammengefasst werden. Lesen mehrere Prozesse dieselbe Pipe, ist unklar, wer die Daten erhält.

`int mkfifo (const char *path, mode_t mode);` Erzeugt eine Pipe mit *Namen* und *Pfad* im Dateisystem. Hat via `chmod` permission bits wie normale Datei. Lebt unabhängig vom erzeugenden Prozess, je nach System auch über Reboots hinweg. Muss explizit mit `unlink` gelöscht werden.

### 8.3. SOCKETS

Ein *Socket repräsentiert einen Endpunkt auf einer Maschine*. Kommunikation findet im Regelfall zwischen zwei Sockets statt (UDP, TCP oder IP sowie Unix-Domain-Sockets). Sockets benötigen für Kommunikation einen Namen: (IP-Adresse, Port)

`int socket(int domain, int type, int protocol);` Erzeugt einen neuen Socket als <Datei>. Socket wird nach Erzeugung zunächst *unbenutzt*. Alle Operationen blockieren per default. `PROT_READ | PROT_WRITE`, // never use execute `MAP_SHARED`, // int flags `fd`, // int file\_descriptor `off_t`, // off\_t offset (start map from first byte) );

**3.9. VERGLEICH MESSAGE-PASSING UND SHARED MEMORY**  
*Shared Memory* ist schneller zu realisieren, aber schwer wartbar. *Message-Passing* erfordert mehr Engineering-Aufwand, schlussendlich aber in Mehr-Prozessor-Systemen bald performanter.

**3.4. VERGLEICH MESSAGE-QUEUES UND PIPES**  
*Message-Queues* *Pipes*  
- bidirektional - unidirektional  
- Daten sind in einzelnen Messages organisiert - übermietet Bytestrom an Daten  
- beliebiger Zugriff - FIFO-Zugriff  
- Haben immer einen Namen - Müssen keinen Namen haben

## 10. UNICODE

**10.1. ASCII - AMERICAN STANDARD CODE FOR INFORMATION INTERCHANGE**  
Hat 128 definierte Zeichen (erste Zeile: *Zeile*, zweite Zeile: *Zeichensatz* - *Spalte*, die 41.,  $\neq$  A).

`send (fd, buf, len, 0) = write (fd, buf, len); new (fd, buf, len, 0) = read (fd, buf, len);` Senden und Empfangen von Daten. Puffern der Daten ist Aufgabe des Netzwerks. `int close (int socket);` Schliesst den Socket für den aufrufenden Prozess. Hat ein anderen Prozess den Socket noch geöffnet, bleibt die Verbindung bestehen. Die Gegenseite wird nicht benachrichtigt. `int shutdown (int socket, int mode);` Schliesst den Socket für alle Prozesse und baut die entsprechende Verbindung ab. Mode: `SHUT_RD` (Keine Les-Zugriffe mehr), `SHUT_WR` (Keine Schreib-Zugriffe mehr), `SHUT_RDWR` (Keine Les- oder Schreib-Zugriffe mehr)

`send (fd, buf, len, 0) = write (fd, buf, len); new (fd, buf, len, 0) = read (fd, buf, len);` Senden und Empfangen von Daten. Puffern der Daten ist Aufgabe des Netzwerks. `int close (int socket);` Schliesst den Socket für den aufrufenden Prozess. Hat ein anderen Prozess den Socket noch geöffnet, bleibt die Verbindung bestehen. Die Gegenseite wird nicht benachrichtigt. `int shutdown (int socket, int mode);` Schliesst den Socket für alle Prozesse und baut die entsprechende Verbindung ab. Mode: `SHUT_RD` (Keine Les-Zugriffe mehr), `SHUT_WR` (Keine Schreib-Zugriffe mehr), `SHUT_RDWR` (Keine Les- oder Schreib-Zugriffe mehr)

**9. MESSAGE PASSING UND SHARED MEMORY**  
Prozesse sind voneinander isoliert, müssen jedoch trotzdem miteinander interagieren. *Message-Passing* ist ein Mechanismus mit zwei Operationen: *Send* (Irgend die Nachricht aus dem Prozess: `send (message);`), *Receive* (Irgend die Nachricht in den Prozess: `receive (message);`). Dabei können Implementierungen nach verschiedenen Kriterien unterschieden werden (*Feste oder Variable Nachrichtengröße*, *direkte oder indirekte / synchrone oder asynchrone Kommunikation*, *puffern*, mit oder ohne Prioritäten für Nachrichten)

**Feste oder variable Nachrichtengröße:** feste Nachrichtengröße ist einfacher zu implementieren, aber umständlicher zu verwenden als variable Nachrichtengröße. **Direkte Kommunikation:** Kommunikation nur zwischen genau zwei Prozess, Sender muss Empfänger kennen. Es gibt *symmetrische direkte Kommunikation* (Empfänger muss Sender auch kennen) und *asymmetrische direkte Kommunikation* (Empfänger muss Sender nicht kennen).

**Indirekte Kommunikation:** Prozess sendet Nachricht an *Mailboxen*, *Ports* oder *Queues*. Empfänger empfängt aus diesem Objekt. Beide Teilnehmer müssen die gleichen Mailbox(en) kennen. **Lebenszyklus Queue:** Wenn diese Queue einen Prozess erzeugt, lebt sie solange wie der Prozess. Wenn sie dem OS gehört, muss das OS das Löschen übernehmen.

**Synchronisation:** Blockierendes Senden: Sender wird solange blockiert, bis die Nachricht vom Empfänger empfangen wird. **Nicht-blockierendes Senden:** (Sender sendet Nachricht und fährt sofort weiter), **Blockierendes Empfangen:** (Empfänger wird blockiert, bis Nachricht verfügbar), **Nicht-blockierendes Empfangen:** (Empfänger erhält Nachricht, wenn verfügbar, oder 0) **Readendezvous:** In Nachrichten und Versand *beide blockierend*, kommt es zum Rendezvous, sobald beide Seiten ihren Aufruf getätigt haben. **Impliziter Synchronisationsmechanismus.**

`// Producer message msg; open (Q); while (1) { // Consumer message msg; open (Q); while (1) { // while (1) { // while (1) { produce_next (&msg); receive (Q, &msg); // blocked until rec. consume_next (&msg); // blocked until sent consume_next (&msg);`

**Pufferung:** *Keine* (Queue-Länge ist 0, Sender muss blockieren), *Beschränkte* (Maximal in Nachrichten, Sender blockiert, wenn Queue voll ist), *Unbeschränkte* (Beliebig viele Nachrichten, Sender blockiert nie). **Prioritäten:** In mehreren Systemen können Nachrichten mit *Prioritäten* versehen werden. Der Empfänger holt die Nachricht mit der *Höchsten Priorität zuerst* aus der Queue.

**9.1.1. POSIX Message-Passing**  
*Message-Queues* mit *variabler Länge*, haben mind. 32 Prioritäten und können *synchron* und *asynchron* verwendet werden.

`msg_t m; open (const char *name, int flags, mode_t mode, struct mq_attr *attr);` Öffnet eine Message-Queue mit systemeigenem name, returnt Message-Queue-Descriptor. (name mit / beginnt, flags und mode wie bei Dateien, mq\_attr: Div. Konfigs & Queue-Status, R/W mit mq\_getattr/mq\_setattr) `int mq_close (mqd_t queue);` Schliesst die Queue mit dem Descriptor `queue` für diesen Prozess. `int mq_unlink (const char *name);` Entfernt die Queue mit dem Namen `name` aus dem System. `int mq_send (mqd_t queue, const char *msg, size_t length, unsigned int priority);` Sendet die Nachricht, die an Adresse `msg` beginnt und Length `length` Bytes lang ist, in die queue. `int mq_receive (mqd_t queue, const char *msg, size_t length, unsigned int *priority);` Kopiert die nächste Nachricht aus der Queue in den Puffer, der an Adresse `msg` beginnt und Length `length` Bytes lang ist. *Blockiert*, wenn die Queue *leer* ist.

### 9.2.1. SHARED MEMORY

Frames des Hauptspeichers werden *zwei (oder mehr) Prozessen P<sub>1</sub> und P<sub>2</sub> zugänglich* gemacht. In  $P_1$  wird Page  $V_1$  auf einen Frame  $F$  abgebildet. In  $P_2$  wird Page  $V_2$  auf denselben Frame  $F$  abgebildet. Beide Prozesse können *beliebig* auf dieselben Daten zugreifen. Im Shared Memory müssen *relative Adressen* verwendet werden.

### 9.2.1. POSIX API

Das OS benötigt eine <Datei>  $S$ , das Informationen über den gemeinsamen Speicher verwaltet und eine *Mapping Table* je Prozess.

`int fd = shm_open ("mysharedmemory", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);` Erzeugt falls nötig und öffnet Shared Memory `mysharedmemory` zum Lesen und Schreiben. `int truncate (int fd, off_t length);` Setzt Größe der <Datei>. Muss *zwingend* nach `shm`.

Erstellung gesetzt werden, um entsprechend viele Frames zu allozieren. Wird für Shared Memory mit ganzzahligen Vielfachen der Page-/Framegröße verwendet.

`int close (int fd);` Schliesst <Datei>. Shared Memory *bleibt aber im System*. `int shm_unlink (const char * name);` Löscht das Shared Memory mit dem Namen. (bleibt vorhanden, solange noch von Prozess geöffnet)

`int mmap (void *address, size_t length);` Entfernt das Mapping. `void * address = mmap ( // maps the shared memory into virt. address space of process 0 // void *virt_address (0 because nobody cares) size_of_shared_memory, // size_t_length (same as used in truncate) PROT_READ | PROT_WRITE, // never use execute MAP_SHARED, // int flags fd, // int file_descriptor off_t offset (start map from first byte) );`

**3.9. VERGLEICH MESSAGE-PASSING UND SHARED MEMORY**  
*Shared Memory* ist schneller zu realisieren, aber schwer wartbar. *Message-Passing* erfordert mehr Engineering-Aufwand, schlussendlich aber in Mehr-Prozessor-Systemen bald performanter.

**3.4. VERGLEICH MESSAGE-QUEUES UND PIPES**  
*Message-Queues* *Pipes*  
- bidirektional - unidirektional  
- Daten sind in einzelnen Messages organisiert - übermietet Bytestrom an Daten  
- beliebiger Zugriff - FIFO-Zugriff  
- Haben immer einen Namen - Müssen keinen Namen haben

## 10. UNICODE

**10.1. ASCII - AMERICAN STANDARD CODE FOR INFORMATION INTERCHANGE**  
Hat 128 definierte Zeichen (erste Zeile: *Zeile*, zweite Zeile: *Zeichensatz* - *Spalte*, die 41.,  $\neq$  A).

`send (fd, buf, len, 0) = write (fd, buf, len); new (fd, buf, len, 0) = read (fd, buf, len);` Senden und Empfangen von Daten. Puffern der Daten ist Aufgabe des Netzwerks. `int close (int socket);` Schliesst den Socket für den aufrufenden Prozess. Hat ein anderen Prozess den Socket noch geöffnet, bleibt die Verbindung bestehen. Die Gegenseite wird nicht benachrichtigt. `int shutdown (int socket, int mode);` Schliesst den Socket für alle Prozesse und baut die entsprechende Verbindung ab. Mode: `SHUT_RD` (Keine Les-Zugriffe mehr), `SHUT_WR` (Keine Schreib-Zugriffe mehr), `SHUT_RDWR` (Keine Les- oder Schreib-Zugriffe mehr)

`send (fd, buf, len, 0) = write (fd, buf, len); new (fd, buf, len, 0) = read (fd, buf, len);` Senden und Empfangen von Daten. Puffern der Daten ist Aufgabe des Netzwerks. `int close (int socket);` Schliesst den Socket für den aufrufenden Prozess. Hat ein anderen Prozess den Socket noch geöffnet, bleibt die Verbindung bestehen. Die Gegenseite wird nicht benachrichtigt. `int shutdown (int socket, int mode);` Schliesst den Socket für alle Prozesse und baut die entsprechende Verbindung ab. Mode: `SHUT_RD` (Keine Les-Zugriffe mehr), `SHUT_WR` (Keine Schreib-Zugriffe mehr), `SHUT_RDWR` (Keine Les- oder Schreib-Zugriffe mehr)

**Codepages:** unabhängige Erweiterungen auf 8 Bit. Jede ist unterschiedlich und nicht erkennbar. **Unicode:** Hat zum Ziel, einen eindeutigen Code für *jedes vorhandene Zeichen* zu definieren. D800, bis DF FF, sind wegen UTF-16 keine gültigen Code-Points.

**Code-Points (CP):** Nummer eines Zeichens - <welches Zeichen?> **Code-Unit (CU):** Einheit, um Zeichen in einem Encoding darzustellen (bietet den Speicherplatz)  $P_i = i$  -tes Bit des unkodierten  $P_i$ ,  $U_i = i$  -tes Code-Unit des kodierten  $P_i$ ,  $B_i = i$  -tes Byte im UTF-8

Jede CU umfasst **32 Bit**, jeder CP kann mit *einer CU* dargestellt werden. Direkte Kopie der Bits in die CU bei Big Endian, bei Little Endian werden  $B_1$  bis  $P_3$  in  $P_3$  kopiert usw. Wird häufig intern in Programmen verwendet. Obere 16 Bits oft <zweckentfremdet>.

**10.2. UTF-32**  
Jede CU umfasst **32 Bit**, jeder CP kann mit *einer CU* dargestellt werden. Direkte Kopie der Bits in die CU bei Big Endian, bei Little Endian werden  $B_1$  bis  $P_3$  in  $P_3$  kopiert usw. Wird häufig intern in Programmen verwendet. Obere 16 Bits oft <zweckentfremdet>.

**10.3. UTF-16**  
Jede CU umfasst **16 Bit**, ein CP benötigt **1 oder 2 CUs**. Encoding muss Endianness berücksichtigen. Die 2 CUs werden *Surrogate* Pair genannt,  $U_{high}$  high surrogate,  $U_{low}$  low surrogate. Bei **2 Bytes (1 CU)** wird direkt gemappt und vorne mit Nullen aufgefüllt. Bei **4 Bytes** sind D800, bis DF FF (Bits 1-21) werden dem Separator *ungültig* und müssen *umgerechnet* werden.

`Encoding UTF-32BE`  
`Encoding UTF-32LE`  
`Encoding UTF-16BE`  
`Encoding UTF-16LE`  
`Encoding UTF-8`  
`Encoding UTF-8`

**Beispiel**  
Encoding von `U+10437 (V)` `001001000010011011`

1. Code-Point  $P$  minus 1 000 00s, rechnen und in Binär umwandeln  
 $P = 10437_{10}$ ,  $Q = 10437_{10} - 10000_{10} = 0437_{10} = 000000001000011011$   
2. Obere & untere 10 Bits in Hex umwandeln  
`00010137`  
3. Oberer Wert mit D8 00s, und unterer Wert mit DC 00s, addieren, um Code-Units zu erhalten  
 $U_1 = 0001 + D800 = 1B01_{16} = 1B01_{16} + DC00_{16}$   
4. R/W/E zusammenhängen  
 $BE = D801D037$ ,  $LE = 01B837D0$

### 10.4. UTF-8

Jede CU umfasst **8 Bit**, ein CP benötigt **1 bis 4 CUs**. Encoding muss Endianness *nicht* berücksichtigen. Standard für Webpages. Etwa Erweiterung von ASCII.

**Code-Point** **U<sub>1</sub>** **U<sub>2</sub>** **U<sub>3</sub>** **U<sub>4</sub>** **signifiant**  
`0s...7Fs` `0xxx xxxxxx` `7` bits  
`80s...7FFs` `110x xxxxxx` `10xx xxxxxx` `11` bits  
`800s...FF Ffs` `1110 xxxxxx` `10xx xxxxxx` `10xx xxxxxx` `16` bits  
`100 00s...10 FF FFfs` `1111 0xxx xx` `10xx xxxxx` `10xx xxxxx` `10xx xxxxx` `21` bits

**10.5. ZAHLEN**  
Beschreiben ein *Intervall physischer konsequenter Blöcke*. Ist 12 Byte groß (48 logische Blocknummern, 68 physikalische Blocknummern, 28 Anzahl Blöcke). Positive Zahlen = Block initialisiert, Negativ = Block voralloziert. Im Inode hat es in den 60 Byte für direkte und indirekte Block-Adressierung Platz für 4 Extents und einen Header.  
**Extent Trees:** *Index-Block* (innerer Knoten des Baums, besteht aus Index-Entry und Index-Block), *Index-Entry* (enthält Nummer des physischen Index-Blocks und kleinste logische Blocknummer aller Kindknoten), *Index-Block* (enthält eigenen Tree-Header und Referenz auf Kindknoten)  
**Extent Tree Header:** besteht aus 4 Extents, weil zusätzlicher Block. Magic Number `FS_MAGIC`, 28, Anzahl Einträge, die direkt auf den Header folgen (28), Anzahl Einträge, die maximal auf den Header folgen können (28), Tiefes des Baums (28) - (Einträge sind Extents, 2:1 Einträge sind Index-Block), Reserviert `gap`  
**Index Nodes:** Spezifiziert einen Block, der *Extents* enthält. Besteht aus einem Header und den Ext-Ents (max. 340 bei 4 KB Blockgröße) Ab 1360 Extents zusätzlicher Block mit Index Nodes nötig.

**Beispiele**  
 $P = E_4 = 000110011010$   
 $\Rightarrow P - P_0 = 00111 = 03_{16}$ ,  $P_1 - P_0 = 100100 = 24_{16}$   
 $\Rightarrow U_4 = C0_{16} = (11000000) + 03_{16} = C3_{16}$ ,  $U_0 = 80_{16} = (10000000) + 24_{16} = A4_{16}$   
 $\Rightarrow A = C3A4$   
 $\Rightarrow P = 1E8F_{16} = 0001111010110111$   
 $\Rightarrow P_1 - P_0 = 01_{16}$ ,  $P_1 - P_0 = 3A_{16}$ ,  $P_2 - P_0 = 37_{16}$   
 $\Rightarrow U_2 = E0_{16} = (11100000) + 01_{16} = E1_{16}$ ,  $U_1 = 80_{16} + 3A_{16} = BA_{16}$ ,  $U_0 = 80_{16} + 37_{16} = B7_{16}$   
 $\Rightarrow A = E1BA7B$

**10.5. ENCODING-BEISPIELE**  
**Zeichen** **Code-Point** **UTF-32BE** **UTF-32LE** **UTF-8** **UTF-16BE** **UTF-16LE**  

A	41s	00000041s	41000000s	41s	0041s	4100s
B	E4s	000000E4s	E4000000s	C3A4s	00E4s	E400s
B	81Bs	00000081Bs	81030000s	CEBAs	0381s	8103s
#	1E8Fs	00001E8Fs	871E0000s	E1BA7Bs	1E8Fs	871E1s
A	10303s	00010303s	30030100s	F09B0C80s	D80D0F30s	00D830DFs

Bei LE / BE werden nur die Zeichen innerhalb eines Code-Points vertauscht, nicht die Code-Points an sich.

**11. EXT2-DATENSYSTEM**  
**Partition** (Ein Teil eines Datenträgers, wird selbst wie ein Datenträger behandelt.), **Volume** (Ein Datenträger oder eine Partition), **Sektor** (Kleinste logische Einheit eines Volumes. Daten werden als Sektoren transferiert. Grösse ist von HW definiert. Enthält Daten und Error-Correction-Codes.), **Format** (Layout der logischen Strukturen auf dem Datenträger; wird vom Dateisystem definiert).

**11.1. BLOCK**  
Ein Block besteht aus *mehreren aufeinanderfolgenden Sektoren* (z.B. 2 KB oder 4 KB normal). Das gesamte Volume ist in *Blöcke aufgeteilt* und Speicher wird *nur in Form von Blöcken* alloziert. Ein Block enthält nur Daten aus *einer einzigen Datei*. Es gibt *Logische Blocknummern* (Blocknummer von Anfang der Datei an gesehen, wenn Datei eine ununterbrochene Abfolge von Blöcken wird) und *Physische Blocknummern* (tatsächliche Blocknummer auf dem Volume).

**11.2. INODES**  
Enthält alle *Metadaten* über die Datei, *ausser Namen* und *off-disk* (Grösse, Anzahl der verwendeten Blöcke, Erzeugungszeit, Löschtage, Modifikationszeit, Löschtage, Owner-ID, Group-ID, Flags, Permissions etc.). Hat eine *fixe Grösse* je Volume: Zweipotenzt, mind. 128 Bytes, max. 1 Block. Der Inode *verweist auf die Blöcke*, die *Daten für diese Datei* enthalten. Enthält ein Array `l_block` mit 15 Einträgen je zu 32 Bit.

**Lokalisierung:** Alle Inodes aller Blockgruppen gelten als *eine grosse Tabelle*. Startet mit 1. **Erzeugung:** Neue Verzeichnisse werden in der Blockgruppe angelegt, die die mit Blockgruppen wird *überschneidungsfrei* *freie Inodes* die *meisten Blöcke frei* hat, Dateien in der Blockgruppe des Verzeichnisses oder nahe Gruppen. Bestimmung anhand *Index-Usage-Bitmaps*. **File-Holes:** Bereiche in der Datei, in der *nur Nullen* stehen. Ein solcher Block wird *nicht alloziert*.

**11.3. BLOCKGRUPPE**  
Eine Blockgruppe besteht aus *mehreren aufeinanderfolgenden Blöcken* bis zu 8 mal der Anzahl Bytes in einem Block.  
**Layout:** Block 0 (Kopie des Superblocks), **Block 1 bis n1** (Kopie der Gruppensuperblockentabelle), **Block n1 + 1** (Block-Usage-Bitmap mit einem Bit je Block der Gruppe), **Block n1 + 2** (Inode-Usage-Bitmap mit einem Bit je Inode der Gruppe), **Block n1 + 3 bis n1 + n1 + 2** (Tabelle aller Inodes in dieser Gruppe), **Block n1 + n1 + 3 bis Ende der Gruppe** (Kopie der eigentlichen Daten). **Superblock:** Enthält alle Metadaten über das Volume (Anzahl, Zeitpunkte, Statusbits, etc. und...) immer an Byte 1024, wegen möglicher Bootdaten vorher. **Sparse Superblock:** Kopien des Superblocks bootden vorher in Blockgruppe 0 und 1, ersten inneren Potenzen von 3, 5 oder 7 gehalten (Sehr hoher Wiederherstellungsgrad, aber deutlich weniger Platzverbrauch). **Gruppensuperblock:** 32 Byte *Beschreibung einer Blockgruppe*. **Superblock:** Enthält alle Metadaten über das Volume (Anzahl, Zeitpunkte, Statusbits, etc. und...) immer an Byte 1024, wegen möglicher Bootdaten vorher. **Sparse Superblock:** Kopien des Superblocks bootden vorher in Blockgruppe 0 und 1, ersten inneren Potenzen von 3, 5 oder 7 gehalten (Sehr hoher Wiederherstellungsgrad, aber deutlich weniger Platzverbrauch). **Gruppensuperblock:** 32 Byte *Beschreibung einer Blockgruppe*. **Superblock:** Enthält alle Metadaten über das Volume (Anzahl, Zeitpunkte, Statusbits, etc. und...) immer an Byte 1024, wegen möglicher Bootdaten vorher. **Sparse Superblock:** Kopien des Superblocks bootden vorher in Blockgruppe 0 und 1, ersten inneren Potenzen von 3, 5 oder 7 gehalten (Sehr hoher Wiederherstellungsgrad, aber deutlich weniger Platzverbrauch). **Gruppensuperblock:** 32 Byte *Beschreibung einer Blockgruppe*. **Superblock:** Enthält alle Metadaten über das Volume (Anzahl, Zeitpunkte, Statusbits, etc. und...) immer an Byte 1024, wegen möglicher Bootdaten vorher. **Sparse Superblock:** Kopien des Superblocks bootden vorher in Blockgruppe 0 und 1, ersten inneren Potenzen von 3, 5 oder 7 gehalten (Sehr hoher Wiederherstellungsgrad, aber deutlich weniger Platzverbrauch). **Gruppensuperblock:** 32 Byte *Beschreibung einer Blockgruppe*. **Superblock:** Enthält alle Metadaten über das Volume (Anzahl, Zeitpunkte, Statusbits, etc. und...) immer an Byte 1024, wegen möglicher Bootdaten vorher. **Sparse Superblock:** Kopien des Superblocks bootden vorher in Blockgruppe 0 und 1, ersten inneren Potenzen von 3, 5 oder 7 gehalten (Sehr hoher Wiederherstellungsgrad, aber deutlich weniger Platzverbrauch). **Gruppensuperblock:** 32 Byte *Beschreibung einer Blockgruppe*. **Superblock:** Enthält alle Metadaten über das Volume (Anzahl, Zeitpunkte, Statusbits, etc. und...) immer an Byte 1024, wegen möglicher Bootdaten vorher. **Sparse Superblock:** Kopien des Superblocks bootden vorher in Blockgruppe 0 und 1, ersten inneren Potenzen von 3, 5 oder 7 gehalten (Sehr hoher Wiederherstellungsgrad, aber deutlich weniger Platzverbrauch). **Gruppensuperblock:** 32 Byte *Besch*