IN2140 – Introduksjon til operativsystemer og datakommunikasjon

Obligatorisk oppgave: Ditt eget filsystem

Vår 2024

1 Introduksjon

I denne obligatoriske oppgaven skal du implementere en skisse av ditt eget filsystem. Vi kaller det en skisse fordi løsningen din selvsagt ikke behøver å omfatte all funksjonalitet som et komplett filsystem bør ha. Hvis du senere tar kurset IN3000/IN4000, Operativsystemer, så vil du implementere en mer avansert filsystemløsning. Mye av det du lærer i denne oppgaven kan komme til nytte. Du gis en struktur inspirert av inoder på Unix-liknende operativsystemer, som OpenBSD, Linux og MacOS. Strukturen inneholder metadata om filer og kataloger, herunder pekere til andre inoder i filsystemet. På denne måten dannes et tre med én rotnode, som vist i eksempelet under.

```
/
├── etc/
│   ├── httpd/
│   │   └── conf/
│   │       └── httpd.conf
│   ├── host.conf
│   └── hostname
└── var/
    └── log/
        └── messages
```

Du skal implementere funksjoner for å arbeide med filsystemet. Hver fil og hver katalog er representert ved en inode, som har en type, et navn, noen andre attributer og informasjon om innhold i filen eller katalogen. For å kunne lese og forandre inoder, så må de hentes fra disken til minnet.

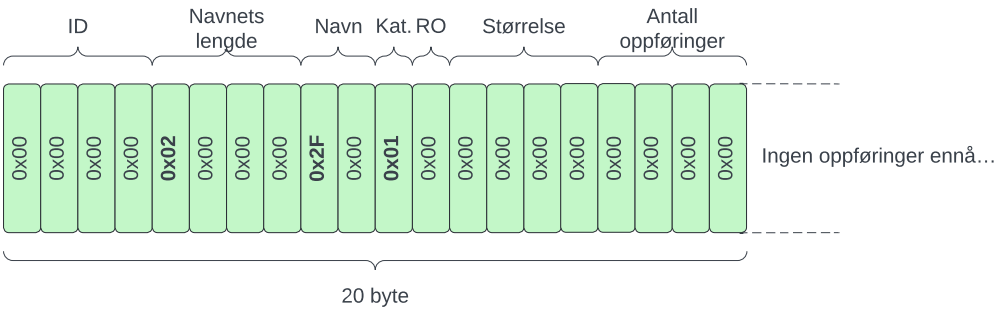
1.1 I minne

Når den ligger i minnet, har en inode følgende struktur:

```
struct inode
{
    int id;
    char*           name;
    char            is_directory;
    int             num_children;
    struct inode** children;
    int             filesize;
    int             num_blocks;
    size_t*         blocks;
};
```

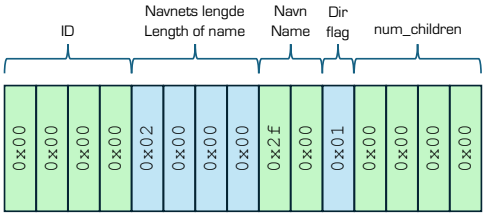
I det følgende forklarer vi feltene i `struct inode`:

- Hver inode har et ID-nummer `id` unikt for hele filsystemet og et fil- eller katalognavn `name`, som inkluderer den endelige null-byten. Rotkatalogen vil alltid ha navnet `"/"`. Vilkårlig lange fil- og katalognavn er tillatt. Lengden på navnet, inkludert den endelige nullbyten, lagres på disken etterfulgt av selve navnet. Lagring av navnelengden på disk gjør arbeidet ditt litt enklere.
- Byteflagget `is_directory` bestemmer om inoden representerer en fil eller en katalog. Dersom `is_directory` er 0, så representerer inoden en fil.
- Hvis `is_directory==1`, representerer inoden en katalog som kan inneholde både andre kataloger eller filer. Antallet filer og kataloger er lagret i `num_children`. Hvis den er 0, er `children` NULL. Ellers inneholder `children` en peker til en dynamisk tildelt rekke inodepekere, som peker til barna. Bare de første `num_children`-pekerne er gyldige.
- Hvis `is_directory==0`, representerer inoden en fil. Denne filen har `filesize`, som gir en simulert filstørrelse i byte. Filen har også `num_blocks`, som er antallet (simulerte) blokker som kreves for å lagre filen med `filesize` byte. Det beregnes ved å runde opp filesize/4096.
- For å spore hvor disse blokkene er lagret på den (simulerte) disken brukes det `blokker`. Dette er en peker til et dynamisk allokert array som er stort nok til å inneholde `num_blocks` `size_t`-verdier som representerer nummeret til en diskblokk. `size_t` er en heltallsverdi uten fortegn, som er 8 byte stor på x86/Linux-datamaskiner på IFI (se `man size_t` for mer informasjon).

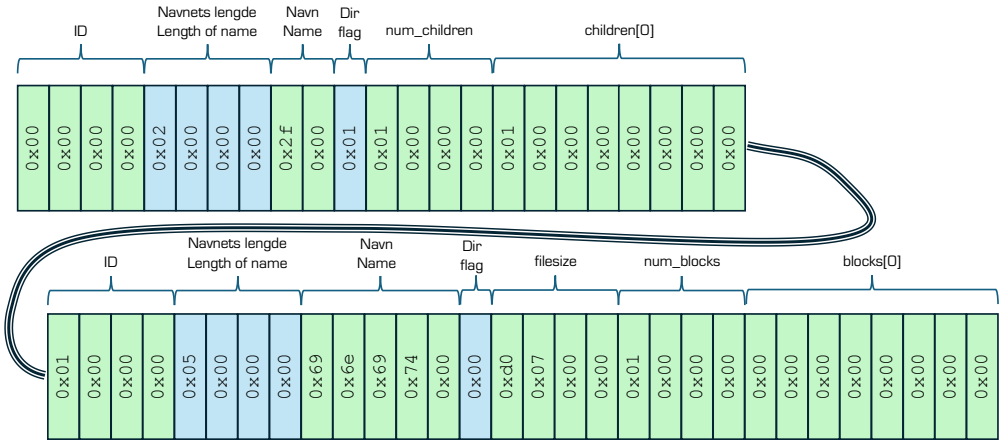


På disk

Ikke alle disse feltene er lagret i master file table på disken. Den viktigste detaljen er at `children` er minneadresser (pekere) som ikke gir mening når de er lagret på disk. Vi



Figur 1: Master file table for et formatert, men tomt filsystem, som bare inneholder rotkatalogen '/'. Hver celle representerer en byte på disken. Merk at heltallsverdiene er lagret i liten endian byte-rekkefølge.



Figur 2: Master file table av et formatert filesystem som inneholder bare katalogen root og filen 'init'.

oppbevarer barnas ID i stedet.

Figur **??** viser hvordan master file table ser ut på en simulert disk som bare inneholder den tomme rotkatalogen. Merk at `num_children` er 0 så lenge ingen andre filer eller kataloger er opprettet. Senere vil hver oppføring fylle 8 byte.

Figur **??** viser en master file table for en simulert disk som inneholder rotkatalogen '/' og en enkel fil kalt 'init'. Du kan se at informasjonen om barn kun lagres for kataloger, og at informasjon om filstørrelse og blokkene som brukes på den simulerte disken kun lagres for filer. Oppføringen `children[0]` inneholder **ID-en** til inoden for 'init' (og ikke en peker til den).

2 Oppgaver

2.1 Design

Skriv en README-fil hvor du forklarer følgende punkter:

- Hvordan du leser *master file table*-filen (som beskrivet på forelesningsfoilene) fra disk og laster inodene inn i minnet.
- Eventuelle implementasjonskrav som ikke er oppfylt.

- Eventuelle deler av implementasjonen som avviker fra prekoden. Dersom du for eksempel oppretter egne filer, forklar hva hensikten deres er.
- Eventuelle tester som feiler og hva du tror årsaken kan være.

2.2 Implementasjon

Vi leverer ut prekode og forventer at dere implementerer de følgende funksjonene, som finnes som skjeletter i filen `inode.c`. Det vil være hensiktsmessig å skrive hjelpefunksjoner i tillegg.

Opprett fil

```
struct inode* create_file(
    struct inode* parent,
    char* name,
    char readonly,
    int size_in_bytes
);
```

Funksjonen tar som parameter en peker til inoden til katalogen som skal inneholde den nye filen. Innenfor denne katalogen må navnet være unikt. Dersom det finnes en fil eller katalog med samme navn der fra før, så skal funksjonen returnere `NULL` uten å gjøre noe.

Opprett katalog

```
struct inode* create_dir(
    struct inode* parent,
    char* name
);
```

Funksjonen tar som parameter inoden til katalogen som skal inneholde den nye katalogen. Innenfor denne katalogen må navnet være unikt. Dersom det finnes en fil eller katalog med samme navn der fra før, så skal funksjonen returnere `NULL` uten å gjøre noe.

Slette en fil

```
int delete_file(
    struct inode* parent,
    struct inode* node
);
```

Funksjonen tar som parametere inodene til en katalog og en fil. Filen, node, skal slettes. Katalogen, parent, inneholder filen. Inodenoden kan slettes av dette kallet bare hvis parent er den direkte foreldrenoden til noden. Operasjonen delete_file mislykkes hvis dette ikke er tilfelle. Hvis det lykkes, returnerer delete_file 0, ellers -1. Blokkene som er allokert på den simulerte disken må frigis ved å bruke funksjonen `free_block`, som er implementert i allocation.c.

Slette en katalog

```
int delete_dir(
    struct inode* parent,
    struct inode* node
);
```

Funksjonen tar som parameter inodene til to kataloger. Den andre, node, er katalogen som skal slettes. Den første, parent, inneholder denne katalogen. Inodenoden kan slettes av dette kallet hvis overordnet er dens direkte overordnede, og hvis katalognoden er tom (ikke inneholder filer eller andre kataloger). Operasjonen delete_dir mislykkes hvis dette ikke er tilfelle. Hvis det lykkes, returnerer delete_dir 0, ellers -1.

Finn inode ved navn

```
struct inode* find_inode_by_name(
    struct inode* parent,
    char* name
);
```

Funksjonen sjekker alle inoder som refereres direkte fra foreldreinoden. Hvis én av dem har navnet `name`, så returnerer funksjonen dens inodepeker. `Parent` må peke på en kataloginode. Dersom ingen slik inode finnes, så returnerer funksjonen `NULL`.

Last filsystem

```
struct inode* load_inodes();
```

Funksjonen leser *master file table*-filen og oppretter en inode i minne for hver tilsvarende oppføring i filen. Funksjonen setter pekere mellom inodene på riktig vis. Superblokkfilen forblir uforandret.

Avslutt filsystem

Skriv følgende funksjon for å koble ned filsystemet.

```
void fs_shutdown(struct inode* node);
```

Funksjonen frigjør alle inode-datastrukturer og alt minne som de refererer til. Dette kan utføres rett før et testprogram avslutter programmet.

2.3 Viktige utleverte funksjoner

```
void save_inodes( const char* master_file_table, struct inode* root )
```

Den utleverte koden inkluderer en funksjon `save_inodes`, som lagrer master file table på disk. Selv om funksjonen `load_inodes` som du må skrive, er vanskeligere fordi du må ta kontekstavhengige beslutninger og må administrere dynamisk minne riktig, kan dette hjelpe deg mye med å forstå hvordan inoder i minnet blir til inoder på disk.

```
void debug_fs(struct inode* node);
```

I prekoden leveres det en funksjon `debug_fs` som skriver ut en inode og hvis denne inoden er en katalog, rekursivt også alle fil- og katalog-inoder under den. Det skrives ID, navn og tilleggsinformasjon.

```
void debug_disk();
```

I tillegg leveres funksjonen `debug_disk`, som skriver ut enten 0 eller 1 for alle blokker på den simulerte disken vår. Har en blokk verdi 1 så finnes det en fil-inode som har reservert blokken til sin fil. Blokker representert av ved 0 er ikke i bruk.

```
int allocate_block();
```

Funksjonen allokerer en diskblokk på 4096 bytes. Man kan ikke allokere mindre enn én diskblokk for fildata, og blokken kan ikke deles mellom filer. Funksjonen returner en blokindeks eller -1 ved ingen ledige blokker. Vi holder denne informasjonen oppdatert på disk i en fil som heter `block_allocation_table`. Dette enkle filsystemet implementerer ikke extent-prinsippet. Man må allokere én blokk á 4096 byte om gangen ved å kalle funksjonen. Vi antar at *master file table* ikke forvaltes på samme måte.

```
int free_block(int block);
```

Når filen slettes, skal diskblokkene frigjøres. Returnerer også -1 ved feil (blokk ikke allokert og så videre).

3 Råd

Her er noen råd som kan gjøre arbeidet med oppgaven litt enklere.

- Det kan være hensiktsmessig å implementere funksjonene i denne rekkefølgen:
  1. Funksjoner for innlasting:
     - `load_inodes`
     - `find_inode_by_name`
  2. Funksjoner som kreves for å lage filer og kataloger og oppdatere *master file table* og *block allocation table*:
     - `create_dir`
     - `create_file`
     - `fs_shutdown`
  3. Funksjoner for a slette filer og kataloger.
     - `delete_file`
     - `delete_dir`
- For å sjekke minnelekkasjer, kjør Valgrind med følgende flagg:

```
valgrind
    --track-origins=yes   \
    --malloc-fill=0x40    \
    --free-fill=0x23      \
    --leak-check=full     \
    --show-leak-kinds=all \
    DITT_PROGRAM
```

4 Levering

1. Legg alle filene i en mappe YOUR_USER_IDS (alle brukere i din gruppe). Du behøver ikke opprette noen nye filer selv, men kan klare deg med filene som du har fått utlevert som prekode (etter at du har implementert funksjonene). Hvis du oppretter egne filer, så husk å kopiere dem inn i mappen også.

```
$ mkdir YOUR_USER_IDS
$ cp -r
        create_example1/ \
        create_example2/ \
        create_example3/ \
        load_example1/   \
        load_example2/   \
        load_example3/   \
        del_example1/    \
        del_example2/    \
        del_example3/    \
        allocation.c     \
        allocation.h     \
        create_fs_1.c    \
        create_fs_2.c    \
```

```
create_fs_3.c    \
load_fs.c        \
del_fs.c         \
inode.c          \
inode.h          \
Makefile         \
YOUR_USER_IDS
```

2. Lag et arkiv som er komprimert med GNU tar-programmet ved å bruke følgende kommando, som du leverer.

   ```
   $ tar czf YOUR_USER_IDS.tar.gz YOUR_USER_IDS/
   ```

3. Sterkt anbefalt! Test innleveringen ved å laste tar.gz-filen ned til en Ifi-maskin. Pakk ut, kompiler og kjør.