IN2140—Introduction to Operating Systems and Data Communications

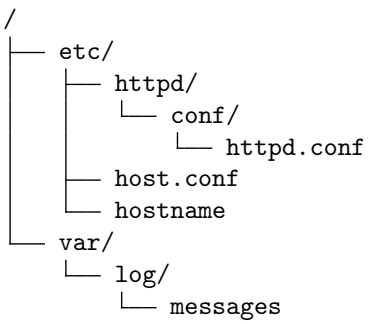
Mandatory Assignment 2: Your Own File System

Spring 2025

Innhold

1 Introduksjon

I denne obligatoriske oppgaven skal du implementere en skisse av ditt eget filsystem. Vi kaller det en skisse fordi løsningen din åpenbart ikke trenger å inkludere all funksjonalitet som et komplett filsystem burde ha. Hvis du senere tar kurset IN3000/IN4000, Operativsystemer, vil du implementere en mer avansert filsystemløsning. Mye av det du lærer i denne oppgaven kan komme godt med. Du får en struktur inspirert av inoder på Unix-lignende operativsystemer, som OpenBSD, Linux og MacOS. De struktur inneholder metadata om filer og directories, inkludert pekere til andre inoder i filen system. På denne måten dannes et tre med én rotnode, som vist i eksempelet nedenfor.



Du må implementere funksjoner for å jobbe med filsystemet. Hver fil og hver directory er representert av en inode, som har en type, et navn, noen andre attributter, og den inneholder nok informasjon for å finne dataene i den filen på selve den fysiske disken. Å kunne lese og endre inoder, må de lastes fra disk til minne. Når den er lagret i minnet, har en inode følgende struktur:

1.1 `struct inode` in memory

```
struct inode
{
    uint32_t   id;
    char*      name;
    char       is_directory;
    char       is_readonly;
    uint32_t   filesize;
    uint32_t   num_entries;
    uintptr_t* entries;
};
```

I det følgende forklarer vi feltene i `struct inode` når det er lastet inn i minnet:

- Hver inode har et ID-nummer `id` unikt for hele filsystemet.
- En inode for en fil eller directory har et navn `name`, som er en peker til en C streng. Generelt kan disse navnene inneholde vanlige tegn og tall, så vel som `'_'` og `'.'`. Vi tillater ikke mellomrom (verken mellomrom eller tabulator). Som en spesielle tilfeller har rotdirectoryet navnet `'/'`. Dette rotdirectoryet eksisterer når disken er nyopprettet (eller formatert). Vilkårlig lange fil- og directorynavn er tillatt.
- Byteflagget `is_directory` bestemmer om inoden representerer en fil eller directory. Hvis `is_directory` er 1, representerer inoden en directory. Hvis `is_directory` er 0, så representerer inoden en fil. Alle andre verdier er feil.
- Et annet byteflagg, `is_readonly` bestemmer om filen eller directoryet som inode representerer er lesbar og skrivbar eller kun lesbar. Det er inkludert som et eksempelattributt, men har ingen praktisk betydning for oppdraget.
- `filesize` gir den simulerte filens størrelse i **bytes**; for directories dette er alltid 0.
- Hver inode inneholder en peker `entries` til en dynamisk allokert matrise med `num_entries` felt på 64 bits. Feltene tolkes forskjellig avhengig av om inoden representerer en fil eller en directory:
  - For filer er hvert 64-bits-felt et par av to usignerte 32-biters heltall **blockno** og **extent**. Blockno tolkes som blokknummer for en blokk på 4096 byte på disk. extent er antall påfølgende blokker på disken som starter med blockno, bare verdiene 1, 2, 3 og 4 er mulige. Det er en sammenheng mellom `filesize` og antall blokker som kreves. Det nødvendige antallet blokker beregnes ved å runde opp filesize/4096.
  - For directories inneholder hvert felt en inodepeker, som representerer en fil eller underdirectory inne i dette directoryet.

1.2 Inode on disk

Inoder må lagres på disken slik at datamaskinen og filsystemet kan slå seg av og starte på nytt senere. Når `struct inode`-er lastes tilbake inn i minnet på et senere tidspunkt, de vil bli opprettet kl. forskjellige adresser i minnet. Det er derfor ikke fornuftig å lagre pekere som `navn` og `entries`.

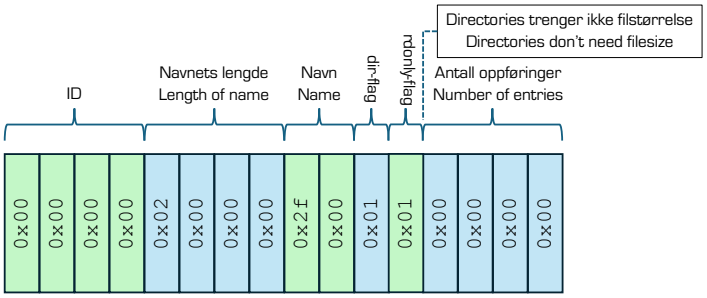
Det er nødvendig å *serialisere* `struct inode`-ene når de skrives til disk.

Her er noen merknader om denne serialiseringen:

- `struct inode`s på disk har forskjellige størrelser, og flere read-operasjoner vil være nødvendig for å lese en enkelt `struct inode` fra disk til minne.
- Lengden på navnet, inkludert den endelige null-byten, lagres på disken etterfulgt av navnet seg selv. Lagring av navnelengden på disk vil gjøre arbeidet litt enklere.
- Hvis en inode er en directory og den er serialisert, skrives `entries` som til disk som en sekvens av 64-biters felt som inneholder `id`ene til `struct inode`s som oppføringen i minnet peker på.
- Hvis en inode er en fil og den er serialisert, skrives `entries` til disk som en sekvens av parer av 32-bits verdier. Verdiene (blockno og extent) er de samme som i minnet.

Figur 1 nedenfor viser hvordan inoden som representerer det tomme rotdirectoryet ser ut når den er på disk. Merk at antall oppføringer er 0 så lenge ingen andre filer eller directories har blitt opprettet. Senere vil hver oppføring fylle 8 byte.

Du kan se at name-feltet inneholder `0x2F` og `0x00`, som er heksadesimal ASCII-verdi for skråstrek '/' etterfulgt av strengavsluttende null.



Figur 1: Master file table for et formatert, men tomt filsystem, som bare inneholder rotdirectoryet '/'. Hver celle representerer en byte på disk. Merk at integerverdiene er lagret i liten endian byte order.

Figur 2 viser toppnivådirectoryet '/' når den inneholder en første fil med navnet 'init'.

Du kan se at inoden for '/' tar mer plass nå. Oppføringen i denne directory-inoden inneholder nå **ID-en** til en annen inode (og ikke en peker til den).

Inoden for 'init' inneholder 4 byte for å lagre filstørrelsen, og den har også én oppføring. Denne oppføringen omfatter to deler, blockno og extent. Når du ser på eksempelet, ser du at filstørrelsen inneholder 0x07d0 i heksadesimal, som er 2000 i desimal.

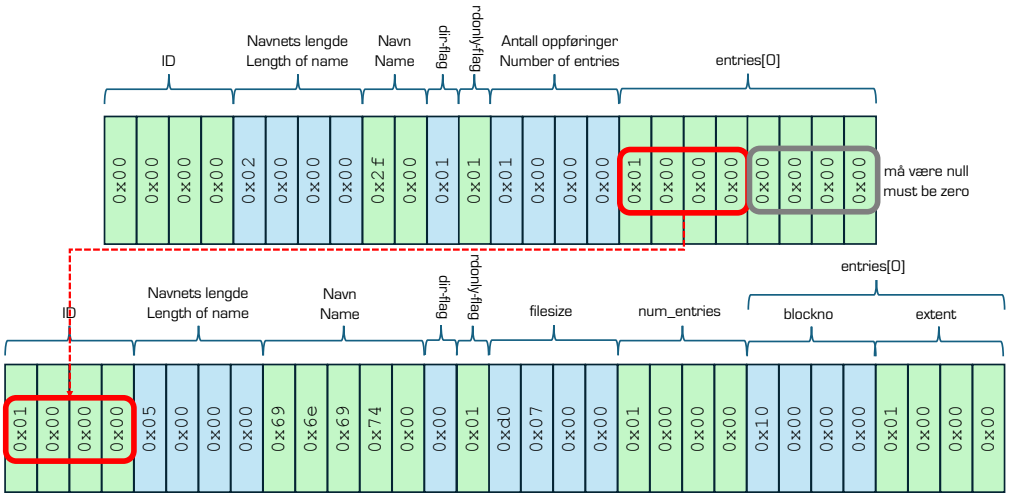
Hver blokk på disken er 4096 byte stor, så 'init' passer inn i en enkelt blokk. Følgelig er det bare én oppføring, det er blokken 0x10 (desimal 16) og extent er 1.

2 Tasks

2.1 Design

Skriv en README-fil som forklarer følgende punkter:

- Hvordan lese masterfiltabellen (som beskrevet i forelesningsbildene) filen fra disken og laste inn inoder inn i minnet.
- Eventuelle implementeringskrav som ikke er oppfylt.



Figur 2: Hovedfiltabell for et formatert filsystem, som inneholder bare rotdirectoryet '/' og filen 'init'.

- Enhver del av implementeringen som avviker fra prekoden. For eksempel hvis du lager dine egne filer, forklar hva deres formål er.
- Eventuelle tester som mislykkes og hva du tror årsaken kan være.

2.2 Implementasjon

Vi gir forhåndskode og forventer at du implementerer følgende funksjoner, som finnes som skjeletter i filen `inode.c`. Det vil være nyttig å skrive hjelpefunksjoner også.

Create File

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

Funksjonen tar som parameter en peker til inoden til directoryet som skal inneholde den nye filen. I dette directoryet må navnet være unikt. Hvis det er en fil eller directory med samme navn der allerede, så skal funksjonen returnere `NULL` uten å gjøre noe.

Parameteren `size_in_bytes` gir antall bytes som må lagres på den simulerte disken for denne filen. Nødvendig antall blokker må tildeles ved hjelp av funksjonen `allocate_block`, som er implementert i allocation.c. Det er mulig at det ikke er det nok plass på den simulerte disken, noe som betyr at et kall til `allocate_block` vil mislykkes. Du bør frigjøre alle ressurser i så fall og returnere `NULL`.

Create Directory

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

Funksjonen tar som parameter inoden til directoryet som skal inneholde det nye directoryet. Innenfor dette directoryet, må navnet være unikt. Hvis det allerede er en fil eller

directory med samme navn der, så skal funksjonen returnere `NULL` uten å gjøre noe.

Find Inode by Name

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

Funksjonen sjekker alle inoder som er referert til *direkte* fra den overordnede inoden. Hvis en av dem har navnet `name`, så returnerer funksjonen inodepekeren som tilbører denne inoden. `parent` må peke til en directory-inode. Hvis ingen slik inode eksisterer, returnerer funksjonen `NULL`.

Delete a file

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

Denne funksjonen sletter en fil referert til av inoden `node` fra det overordnede directoryet referert til med inoden `parent`.

Funksjonen kaller `free_block` for hver blokk som refereres til av denne filen. Dette gjelder også for extent: hvis extent har lengden 4, må `free_block` kalles 4 ganger. Dette fjerner disse blokkene fra simuleringsdisken.

Denne funksjonen gjør ikke noe og returnerer en feilmelding hvis `node` ikke er en fil, hvis `parent` er ikke en directory, eller hvis `parent` ikke er directoryet som inneholder `node`.

Returnerer 0 ved suksess og -1 ved feil.

Delete a directory

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

Denne funksjonen sletter en tom directory referert til av inoden `node` fra dens overordnede directory referert av sin inode `parent`. Den frigjør alt minne knyttet til `node`.

Denne funksjonen gjør ikke noe og returnerer en feilmelding hvis `node` ikke er en directory, hvis `parent` er ikke en directory, eller hvis `parent` ikke er directoryet som inneholder `node`. Den gjør heller ingenting og returnerer en feil hvis `node` er det en directory, men er ikke tom, dvs. inneholder fortsatt andre inoder.

Returnerer 0 ved suksess og -1 ved feil.

Save File System

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

Funksjonene skriver den gitte inoderoten og alle inodene referert av den til masterfiltabellen, følge de forpliktende instruksjonene. Ingen inoder endres.

Load File System

```
struct inode* load_inodes( const char* master_file_table );
```

Funksjonen leser den gitte masterfiltabellfilen og lager en inode i minnet for hver tilsvarende oppføring i filen. Funksjonen plasserer pekere mellom inodene riktig. Hovedfiltabellfilen forblir uendret.

Returnerer en peker til rot-inoden ved suksess og `NULL` ved feil.

Shut Down File System

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

Funksjonen frigjør alle inodedatastrukturer og alt minne de refererer til. Dette bør gjøres umiddelbart før et testprogram avslutter programmet.

2.3 Diskusjon med evaluatoren

Under evalueringen av den obligatoriske oppgaven vil evaluatorer tilfeldig velge ut grupper som må være forberedt å møte sin evaluator for en diskusjon.

En gruppe som blir bedt av sin evaluator om å møtes for en slik diskusjon må gjøre det, det er ikke valgfritt eller åpen for forhandlinger. Diskusjonene vil være ganske korte og bør helst skje i en av gruppetimine. Evaluatoren må være trygg på at hvert gruppemedlem har forstått koden som du har levert.

Invitasjonen til diskusjon kan komme om evaluator mener at en løsning er klar til å bestå uten ytterligere endringer eller ikke. Å gå bredt vil hjelpe oss til å forstå bedre hvor klare kursdeltakerne er for hjemmeeksamen.

Diskusjonen vil vanligvis erstatte den skriftlige tilbakemeldingen til gruppene som inviteres til diskusjon (selv om evaluator har lov til å skrive også), så vi anbefaler gruppene å ta notater nøye.

2.4 Forventninger

Den obligatoriske oppgaven kan bestås når alle eksempler som kun leser master file table og block allocation table fungerer feilfritt, noe som betyr at det er mulig å stå uten noen forsøk til å løse operasjonene som endrer inode-treet i minnet eller på disk.

Imidlertid tolker vi feilfritt veldig strikt her: ingen kompilatorvarsel etter å ha laget en nyopprettet build-katalog, ingen warnings ved kjøretid, ingen warnings eller feil fra valgrind, ingen krasj, og forventet output matcher perfekt for testene vi deler ut, så vel som for tilleggstestene vi ikke deler ut.

For å kompensere for mangler, må du gjøre fremskritt mot riktig arbeid av modifikasjons- og skriveoperasjoner også.

3 Flere detaljer

3.1 Viktige utlverte funksjoner

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

Prekoden leverer en funksjon `debug_fs` som skriver ut en inode og hvis denne inoden er en directory, rekursivt også alle fil- og directory-inoder under den. ID, navn og tilleggsinformasjon er skrevet.

Hvis denne funksjonen krasjer eller valgrind oppdager lesing utenfor grensen når du prøver å feilsøke inodene dine, er det svært sannsynlig at du ikke har implementert inoder som tiltenkt med denne oppgaven.

```
void debug_disk();
```

`debug_disk` skriver ut flere linjer med 0-er og 1-er. Hver 0 eller 1 representerer en blokk på disk, som starter med blokk 0. Hvis en blokk har verdi 1, er det en filinode som har reservert blokk for filen. Blokker representert ved 0 er ikke i bruk.

Hvis du (for eksempel) mistenker feil ved tildeling og frigjøring av extents, kan du sammenligne blokknumrene lagret i inodene dine med blokknumrene vist som tildelt av denne funksjonen.

```
int allocate_blocks( int extent_size );
```

Funksjonen prøver å tildele opptil `extent_size` påfølgende diskblokker, hvor hver blokk kan rommer 4096 byte. Du kan ikke tildele mindre enn én diskblokk for en fil, og en blokk kan ikke deles mellom filer. Funksjonen returnerer blokkindeksen til den første blokken i extenten eller -1 if forespørselen kan ikke oppfylles fullstendig. Hvis en tildeling mislykkes, bør du prøve på nytt med en mindre `extent_size`. Det vil alltid mislykkes når $extent\_size <= 0$ eller $extent\_size > 4$. Vi beholder denne informasjon oppdatert på disken i en fil kalt `block_allocation_table`.

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

Når filen er slettet, må diskblokkene frigjøres. Det er viktig at hver enkelt blokk av en extent må frigis separat fordi `block_allocation_table` ikke vet noe om extenten. `free_block` returnerer -1 i tilfelle feil (blokk ikke tildelt og så videre).

3.2 Råd

Her er noen råd for å hjelpe deg i gang.

- Det kan være hensiktsmessig å implementere funksjonene i denne rekkefølgen:
  1. `load_inodes` og `find_inode_by_name`
  2. `create_dir` og `create_file`, alloker kun extents av størrelse 1 på dette tidspunktet
  3. `fs_shutdown`
  4. `delete_dir` og `delete_file`
  5. legger til støtte for størrelser 2, 3 og 4 i `create_file` og `delete_file`
- 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 \
    YOUR_PROGRAM
```

3.3 Compilation

For å kompilere programmene har vi laget en spesifikasjonsfil kalt CMakeLists.txt igjen.

Du kan laste ned og kompilere dette prosjektet på følgende måte: Åpne en terminal.

- Pakk ut de medfølgende filene i terminalen:

```
tar zxvf oblig-02.tgz
```

- gå inn i directory oblig-02

  ```
  cd oblig-02
  ```

- opprette en underdirectory for å bygge programmene fra den oppgitte kildekoden

  ```
  mkdir bygge
  ```

- gå inn i byggedirectoryet

  ```
  cd bygge
  ```

- bruk CMake for å lage en Makefil som passer din maskin

  ```
  cmake ..
  ```

- kompilere for å lage kjørbare filer fra kilekoden

  ```
  make
  ```

- kjøre programmene, f.eks.

  ```
  create_example1
  ```

3.4 Testing

For å hjelpe deg med å komme i gang utleverer vi flere testprogrammer som tester et subsett av funksjonene du må implementere. Testene er følgende:

- `check_disk`: Dette er kun test blokkallokeringsfunksjonene som har blitt delt ut. Hvis utdata fra denne testen ikke er det samme som forventet-utdata/test-1-1-expected-output.txt, noe er feil med oppsettet ditt.
  MERK: Når vi sier 'det samme', mener vi ikke bokstavelig talt det samme, fordi test-outputen inneholder prosessnumre som alltid er forskjellige, samt testerens hjemmedirectory, som er forskjellig for hver av oss. Men resultatet av selve programmet, så vel som testresultatene fra valgrind skal være det samme.

- `check_fs`: Dette laster den gitte master file table og block allocation table inn i minne, skriver ut directorytreet med tilleggsinformasjon for hver inode, skriver ut et kart over blokkene som er tildelt i henhold til informasjonen som er lagret i inodene, og den skriver ut et kart over blokkene som faktisk er allokert i henhold til block allocation table. Forventningsresultatene er i forventet-outputs/test-2-*-expected-output.txt

- `load_fs_1`, `load_fs_2` og `load_fs_3`: Starter som `check_fs`, men er ment å laste spesifikke eksempeltabeller. Prøver å finne spesifikke filer og directories i de innlastede filsystemene. De forventede resultatene er i forventet-outputs/test-3-*-expected-output.txt

- `create_fs_1`, `create_fs_2` og `create_fs_3`: Starter med tom hovedfiltabell og blokkallokeringstabell som er gitt på kommandolinjen, formaterer disken, og lager filer og directories er spesifisert i kildekoden til disse 3 testkommandoer. De forventede resultatene er i forventet-output/test-4-*-expected-output.txt

- `create_and_delete_test`: Liker `create_fs_*`, men prøver i tillegg å utføre noen slettinger av filer og directories som er ment for å mislykkes, og noen slettinger som er ment å lykkes. De forventede resultatene er i forventet-outputs/test-5-1-expected-output.txt

4 Delivery

Du bør sende inn gruppens kode til `https://devilry.ifi.uio.no/` som en arkivfil. Arkivfilen skal være en Zip-fil og må inneholde alle kildefiler og dokumentasjonsfiler.

Følger du 'make'-oppskriften under Compilation, kan du opprette arkivfilen ved å utføre

```
make package_source
```

Filen heter da Oblig02-1.0-Source.zip.

Det anbefales på det sterkeste at du tester bidraget ditt før du laster det opp! Test innsendingen ved å laste ned tar.gz-filen til en Ifi-maskin. Pakk ut, kompiler og kjør.