Project 1: https://github.com/jepuli124/systemProgramminCourse/tree/main/project1

Tekijät: Lauri Heiskanen, Konsta Jalkanen

Reverse

Reverse ohjelma käy getline() function avulla tiedosto rivi kerrallaan rekursiivisesti aiheuttaen että ulostuleva teksti on riveittäin käänteisessä järjestyksessä.

```c
void reverse_file(FILE *file_out, FILE *file_in, int argc) { // reverses file
    char *string = NULL;
    size_t n = 0;

    if (argc == 1 && getline(&string, &n, file_in) > 1) { // recursivly calls it self and takes a line to output causing lines appearing in reverse order.
        reverse_file(file_out, file_in, argc);
        fprintf(file_out, "%s", string);
    }
    else if (argc > 1 && getline(&string, &n, file_in) != -1) {
        reverse_file(file_out, file_in, argc);
        fprintf(file_out, "%s", string);
    }
    free(string);
    return;
}
```

Komento syntax: prompt> ./reverse file.txt file2.txt

Onko tiedostot samoja (hardlink) tarkastus: https://stackoverflow.com/questions/12502552/can-i-check-if-two-file-or-file-descriptor-numbers-refer-to-the-same-file

Project 2: https://github.com/jepuli124/systemProgramminCourse/tree/main/project2
Tekijät: Lauri Heiskanen, Konsta Jalkanen

My-cat

My-cat käy jokaisen annetun tiedoston läpi siten, että se lukee ja tulostaa tiedoston rivi kerrallaan. Lukemiseen käytetään *getline*-funktiota, joka itse varaa luetulle merkkijonolle tilan muistista. Kun rivi on tulostettu, *getline*:n varaama muisti vapautetaan.

```
while (getline(&line, &n, file_in) > 0) {
    // print line
    fprintf(file_out, "%s", line);
    free(line);
    line = NULL;
}
```

line (rivin alun osoite) ja *n* (rivin pituus) alustetaan arvoiksi *NULL* ja 0, jotta *getline* varaa itse tarvitun muistin. Tulostamisen jälkeen *line* asetetaan uudestaan arvoon *NULL*.

```
char *line = NULL;
size_t n = 0;
```

Komento syntax: ./my-cat file.txt

My-grep

My-grep toimii hyvin samalla tavalla kuin my-cat. Ainoa varsinainen ero on se, että my-grep tarkistaa löytyykö hakutermi riviltä ennen rivin tulostusta.

```
if (search_line_for_term(line, search_term)) {
    fprintf(file_out, "%s", line);
}
```

Tarkistaminen tapahtuu naivilla algoritmilla, joka tutkii merkkijonoa kahden indeksin avulla.

Indeksi *i* on tutkittavan tekstialueen ensimmäinen merkki. Käytännössä se merkkaa sitä kohtaa rivissä, mistä epäillään alkavan haluttu merkkijono.

Indeksi *j* on etsittävän merkin indeksi hakutermissä. Sillä tutkintaan, onko *j*:des merkki *i*:stä sama merkki kuin *j*:des merkki etsityssä merkkijonossa.

```
(line[i + j] == search_term[j])
```

Jos merkit eivät täsmää, *i*:tä kasvatetaan yhdellä ja *j* asetetaan nollaan. Hakua jatketaan, kunnes rivi loppuu tai hakutermi löytyy.

Komento syntax: ./my-grep hakutermi [file.txt file2.txt …]

My-zip

My-zip avaa ja käsittelee annetut tiedostot yksitellen:

```c
for (size_t i = 1; i < argc; i++) {
    FILE *file;
    // try to open file
    if ((file = fopen(argv[i], "r"))){
        compress_file(file, &prev_char, &length);
    } else {
        // if opening fails, exit
        fclose(file);
        file = NULL;
        exit(1);
    }
    fclose(file);
    file = NULL;
```

Zipatessa luetaan sisään tuleva merkki ja jos se on sama kuin edellinen niin lisätään laskuria yhdellä, ja jos se on eri, niin kirjoitetaan ulostuloon luku paljon merkkejä on ja kyseinen merkki:

```c
void compress_file(FILE *file, char *prev_char, __int32_t *length) {


    // while not at end-of-file
    while (!feof(file)) {
        // read character
        char c = fgetc(file);

        // break if this is the last character (terminator)
        if (feof(file)) {
            break;
        }

        // check if the character if the same as previous
        // or the previous character is the default
        if ((c == *prev_char || *prev_char == '\0')) {
            // add 1 to length if it is
            (*length)++;
        } else {
            // write if it is not
            fwrite(length, sizeof(__int32_t), 1, stdout);
            fwrite(prev_char, sizeof(char), 1, stdout);
            *length = 1;
        }

        // set previous character
        *prev_char = c;
    }
}
```

Luvun tallentamiseen käytetään neljä tavua ja merkin tallentamiseen yksi.

Komento syntax: prompt> ./my-zip file.txt [file2.txt …]

My-unzip

My-zip avaa ja käsittelee annetut tiedostot yksitellen. Unzipatessa luetaan sisään tulevat 5 tavua:

```c
void decompress_file(FILE *file) {
    data_t buffer;
    // read data into buffer
    while((fread(&buffer, sizeof(data_t), 1, file))){
        // print the given character the given number of times
        for (size_t i = 0; i < buffer.n; i++) {
            printf("%c", buffer.c);
        }
    }
}
```

ja tallennetaan se structiin data_t:

```c
// this has to packed so it is exactly 5 bytes
typedef struct {
    __int32_t n;
    char c;
} __attribute__((packed)) data_t;
```

. Sen jälkeen structiin tallennettu viimeinen tavu tulostetaan stdout:iin ensimmäisen 4 tavuun tallennetun luvun verran.

Komento syntax: prompt> ./my-unzip file.txt [file2.txt …]

Project 5: https://github.com/jepuli124/systemProgramminCourse/tree/main/project5
Tekijät: Lauri Heiskanen, Konsta Jalkanen

Pzip
Pzip ohjelma alkaa sillä että tiedosto puretaan suureen memory mappiin:

```c
for (size_t i = 1; i < argc; i++) {
    if ((fp = fopen(argv[i], "r"))){ //opens a file then makes a memory map with mmap, then adds it to overal memory map
        file_length += get_file_length(argv[i]);
        currentSize += file_length;
        char * tempfm = mmap(NULL, file_length, PROT_READ, MAP_PRIVATE, fileno(fp), 0);
        fm = realloc(fm, sizeof(char)*currentSize);
        strcat(fm, tempfm);
        fclose(fp);
        fp = NULL;
        tempfm = NULL;

    } else {
        // if opening fails, exit
        printf("failed opening file %s\n", argv[i]);
        exit(1);
    }
}
```

, joka myöhemmin jaetaan threadejen kesken käsiteltäväksi. Jokainen threadi muodostaa oman tulostettavan osionsa:

```c
void *p_compress(void *pthread_args) {  // goes file symbol by symbol and counts them, then returns the chars and amounts
    pthread_arg_t *args = (pthread_arg_t*) pthread_args;

    pthread_ret_t *ret = malloc(sizeof *ret);
    ret->data = malloc(sizeof(data_t) * (args->end_index - args->start_index));

    // begin zipping
    char prev_char = args->fm[args->start_index];
    __int32_t length = 0;
    size_t file_index = args->start_index;
    size_t data_index = 0;

    while (file_index < args->end_index) {
        // check if the character if the same as previous
        if ((args->fm[file_index] == prev_char)) {
            // add 1 to length if it is
            length++;
        } else {
            // write to data if it is not
            ret->data[data_index].n = length;
            ret->data[data_index].c = prev_char;
            data_index++;
            length = 1;
        }
        // set previous character
        prev_char = args->fm[file_index];
        file_index++;
    }

    ret->data[data_index].n = length;
    ret->data[data_index].c = prev_char;
    ret->last_used_index = data_index;

    pthread_exit(ret);
}
```

, jotka sitten läpikäydään toisissa threadeissa, jotka hoitavat kirjoittamisen ulostuloon:

```c
void *write_chunk(void *args) {
    pthread_ret_t *pret = (pthread_ret_t *) args;
    fwrite(pret->data, sizeof(data_t), pret->last_used_index + 1, stdout);

    pthread_exit(NULL);
}
```

Komento syntax: prompt> ./pzip file.txt [file2.txt …]

Punzip

Jokainen annettu tiedosto avataan erikseen ja siitä tehdään memory mappi:

```
for (size_t counter = 0; counter < file_length; counter += threadLimit * 5)
{
    // each thread handles single characater
    size_t num_of_threads = 0;
    if (counter + (threadLimit * 5) > file_length) {
        num_of_threads = (size_t) floor((file_length - counter)/5);
    } else {
        num_of_threads = threadLimit;
    }
    void ** returnValues = malloc(sizeof(char *) * (num_of_threads));


    pthread_arg_t * p_args = malloc(sizeof(pthread_arg_t) * (num_of_threads));
```

Memory mappia käsittellään 5 tavua kerrallaan jokaisella threadi ajolla:

```
for (size_t counter = 0; counter < file_length; counter += threadLimit * 5)
{
    // each thread handles single characater
    size_t num_of_threads = 0;
    if (counter + (threadLimit * 5) > file_length) {
        num_of_threads = (size_t) floor((file_length - counter)/5);
    } else {
        num_of_threads = threadLimit;
    }
    void ** returnValues = malloc(sizeof(char *) * (num_of_threads));


    pthread_arg_t * p_args = malloc(sizeof(pthread_arg_t) * (num_of_threads));
```

. Threadiin annetut 5 tavua käsitellään niin että neljä ensimmäistä ovat lukuja ja viimeinen kirjain:

```
oid decompress_file(pthread_arg_t *p_args, char ** returnValue) {
   data_t buffer;

   char * text = malloc(sizeof(char)*2); //allocating start memory
   text[0] = '\0';
   // read data into buffer by byte at time
   buffer.n = (p_args->fm[p_args->start_index+3] << 24) | (p_args->fm[p_args->start_index+2] << 16) | (p_args->fm[p_args->start_index+1] << 8) | (p_args->fm[p_args->start_index
   buffer.c = p_args->fm[p_args->start_index+4];


   // print the given character the given number of times
   for (size_t i = 0; i < buffer.n; i++) {
       text[strlen(text)+1] = '\0'; // adding end of text mark to prevent random symbols from new memory to spawn
       if ((text = realloc(text, sizeof(char)*(strlen(text)+5))) == NULL) { //allocating correct amount of memory
           perror("realloc didn't succeed \n");
           }
       text[strlen(text)] = buffer.c;
   }

   //printf("inside decompress file:\n %s\n", text);
   text[strlen(text)+1] = '\0'; //adding end mark
   *returnValue = text;
   text = NULL;
   return;
```

Threadi muodostaa näistä merkijonon joka sitten kirjoitetaan threadeillä myöhemmin:

```c
for (size_t i = 0; i < num_of_threads; i++) { //printing out values
    printf("%s", (char *) returnValues[i]);

}
```

Komento syntax: prompt> ./punzip file.txt [file2.txt …]

threadi komennon käyttäminen:
https://www.geeksforgeeks.org/multithreading-in-c/