Käyttöjärjestelmät ja Systeemiohjelmointi harjoitustyö dokumentaatio

Aarre Urtamo

July 2024

Sisällys

1 Osa 1: Warmup to C and Unix programming

Projektin ensimmäisen osan ideana on rakentaa *reverse*-ohjelma, joka kääntää sille annetun tiedoston rivien järjestyksen käänteiseksi. Ohjelmaa on pystyttävä ajamaan kolmella eri tavalla riippuen *input*-parametrien määrästä. Jos ohjelmaa kutsutaan ohjelman nimen lisäksi yhdellä parametrila, niin ohjelma tulkitsee tämän parametrin *input*-tiedostoksi, jonka rivit käännetään. Tällöin ohjelma tulostaa käännetyn version *standard output*:iin. Jos taas annetaan kaksi parametria, niin ohjelma tulkitsee toisen parametrin *output*-tiedostoksi ja kirjoittaa siihen käännetyn version *standard output*:in sijaan. Kolmas vaihtoehto on, että ei anna ohjelmalle yhtään parametria itse komennon nimen lisäksi. Tällöin ohjelma ottaa käännetävän tiedoston *standard input*:sta ja tulostaa käännetyn version *standard output*:iin.

Aloitin projektin tekemisen muistelemalla miten linkitetty lista implementoidaan C-ohjelmoinnissa. Suunnitelmanani oli tallentaa rivit linkitettyyn listaan, jonka avulla voin kirjoittaa rivit käänteisessä järjestyksessä. Aluksi ohjelmoin ihan perus linkietyn listan, joka tallentaa numerot 1-10. Tämän jälkeen ohjelmoin linkitetyn listan, johon tallensin jokaiseen alkioon tekstiä. En ollut aikaisemmin tehnyt linkitettyä listaa, jossa alkiot pystyi käymään läpi myös käännteisessä järjestyksessä, joten seuraavaksi harjoittelin tekemään niin. Lisäsin jokaiseen solmuun osoittimen myös solmua edeltävään solmuun, kun normaalisti olin aina laittanut ainoastaan osoittimen solmusta seuraavaan solmuun.

```c
// Määritetään tietue linkitettyä listaa varten
typedef struct yksiRivi {
    char *pRivi;
    struct yksiRivi *pSeuraava;
    struct yksiRivi *pEdellinen;
} YKSIRIVI;
```

Kuva 1: Linkitetyn listan solmun rakenne

Kun olin saanut linkitetyn listan toimimaan, aloitin itse ohjelman tekemisen. Aluksi tutustuin *getline*-funktioon. Ensimmäinen versio ohjelmastani toimi niin, että ajaessa tuli antaa yksi parametri, eli ohjelma luki käännetävän tiedoston tästä parametrista ja tulosti käännetyn version *standard output*:iin. Kun olin saanut tämän ensimmäisen version toimimaan, tein omat aliohjelmat kaikille kolmelle eri tavalle, jolla ohjelmaa voidaan ajaa. Pääohjelmaan jätin siis vaan $if - else$-rakenteen, jolla valitaan ohjelman ajotapa parametrian määrää katsomalla. Lopuksi lisäsin ohjelmaan virheenkäsittelyn sekä kommentoinnin.

```c
int main(int argc, char *argv[]) {

    // Tarkistetaan komentoriviparamterien määrä ja else if-rakennetta käyttäen, mitä aliohjelmaa käytetään.
    if (argc > 3) {
        fprintf(stderr, "usage: reverse <input> <output>\n");
        exit(1);
    } else if (argc == 3) {
        parametreja3(argv);
    } else if (argc == 2) {
        parametreja2(argv);
    } else if (argc == 1) {
        parametreja1(argv);
    }
    return(0);
}
```

Kuva 2: Pääohjelman $if - else$-rakenne

Ohjelmaani tuli ylimääräistä koodia, koska en vielä ohjelmaa aloittaessa muistanut kuinka aliohjelmat C-ohjelmoinnissa toimivat. En käyttänyt energiaa ohjelman optimoimiseen vaan itse ohjelman välttämättömään toiminnallisuuteen. Kun olin ohjelmoinut projektin kolmannen osan, C-ohjelmointi oli tullut taas tutummaksi. Huomasin, että olisin voinut tehdä tämän ensimmäisen osan eri tavalla enemmän aliohjelmia hyödyntäen. Ohelma kuitenkin toimii ja tässä harjoitustyössä on huomattavissa kehityskaari C-ohjelmointitaidoissani.

2 Osa 2: Unix Utilities

Projektin toisessa osassa ideana on ohjelmoida neljä pelkistettyä versioita Unixin peruskomennoista. Ensimmäinen komennoista on $my - cat$-komento, joka tulostaa sille parametriksi annetun tiedoston. Toisena on $my - grep$-komento, joka etsii merkkijonoa tai merkkiä tiedostosta ja tulostaa tiedoston rivit, joilta tämä löytyi. Kolmas komento $my - zip$ ja neljäs komento $my - unzip$ pakkaavat ja purkavat tiedoston käyttäen RLE koodausta ja ASCII tunnisteita.

Aloitin ensin tekemään $my - cat$-komentoa. Komentoa ajaessa voidaan antaa komennon nimen lisäksi kuinka monta parametria tahansa. Nämä parametrit tulkitaan kaikki omiksi tiedostoikseen, jotka vain tulostetaan peräkkäin siinä järjestyksessä kuin ne on annettu parametreiksi. Aloitin kuitenkin ohjelmoimalla komennosta version joka lukee vain yhden tiedoston. Luin rivit tiedostosta linkitettyyn listaan. Tämän jälkeen tulostin rivit linkitetystä listasta. Kun olin saanut ohjelman toimimaan yhdellä tiedostolla, tein *while*-loopin, jossa luetaan ja tulostetaan uusi tiedosto jokaisella iteraatiolla, kunnes iteraatioiden määrä ylittää komennon ajossa annetujen parametrien määrän. Lopuksi tein virheenkäsittelyn ja kommentoin koodin.

Seuraavaksi tein $my-grep$-komennon. Komento toimii, kun sille antaa ajaessa ainakin yhden parametrin komennon nimen lisäksi. Tämä ensimmäinen parametri on aina merkkijono, jota etsitään. Jos antaa ainoastaan tämän ensimmäisen parametrin, niin $my-grep$-komento etsii annettua merkkijonoa standard-input:sta. Toteutin komennon, niin että my-grep lukee tällöin *standard input*:ista

kaikki siihen syötetyt rivit, ja vasta tämän jälkeen kirjoittaa rivit, joissa kyseinen merkkijono oli. Toteutin tämän linkitetyn listan avulla. Jos parametreja antaa enemmän kuin kaksi (nimi, merkkijono), niin loput parametrit tulkitaan tiedostoiksi, joista merkkijonoa etsitään. Tein tiedostojen lukemista varten samankaltaisen *while*-loopin kuin my-cat-komennossa. Luin jokaisella iteraatiolla aina uuden tiedoston ja etsin siitä merkkijonoa. Käytin ohjelmassa merkkijonon etsimiseen *strstr*-komentoa. Lopuksi tein koodiin virheenkäsittelyn ja kommentoinnin.

```
// Kirjoitetaan hyväksytyt rivit stdout:iin
while (merkitRivilla > 0) {

    // Tallennetaan ensin hyväksytyt rivit linkitettyyn listaan.
    if (strstr(rivi,argv[1]) != NULL) {

        if ((pUusi = (SOLMU*)malloc(sizeof(SOLMU))) == NULL) {
            fprintf(stderr, "malloc failed\n");
            free(rivi);
        }

        pUusi->pRivi = strdup(rivi);
        pUusi->pSeuraava = NULL;
        if (pAlku == NULL) {
            pAlku = pUusi;
            pLoppu = pUusi;
        } else {
            pLoppu->pSeuraava = pUusi;
            pLoppu = pUusi;
        }
    }
    merkitRivilla = getline(&rivi,&pituus,stdin);
}

// Tulostetaan sitten hyväksytyt rivit linkitetystä listasta
```

Kuva 3: Merkkijonon sisältävien rivin tallentaminen linkitettyyn listaan yhden parametrin tapauksessa.

Kolmantena tein *my−zip*-komennon. Aloitin komennon ohjelmoimisen muuttamalla merkkijonon "aaabb"muotoon "3a2b". Kun sain tämän toimimaan, yleistin koodini kaikille merkkijonoille. Tein *while*-loopin, jossa käydään läpi kaikki tiedoston merkit. Tein muuttujan *luku*, joka kertoi, kuinka monta kyseistä merkkiä oli peräkkäin. Tein merkkejä varten kaksi muuttujaa: *vanha*, *uusi*. Muuttujaan *uusi* tallensin aina, juuri tiedostosta luetun merkin ja muuttujaan *vanha* edellisen luetun merkin. Jos muuttujat *vanha* ja *uusi* olivat samat korotin *luku*-muuttujaa yhdellä. Jos taas muuttujat olivat erit, niin tulostin *luku*-muuttujan ja *vanha* muuttujaa vastaavan ASCII arvon *standard output*:iin. Kun olin saanut ohjelman toimimaan yhdellä tiedostolla, niin tein *for*-loopin, jossa käydään niin monta iteraatoita läpi kuin komennon ajossa annettiin parametreja. Lopuksi tein ohjelmaan virheenkäsittelyn muistinvarauksiin sekä tiedoston avauksiin.

```
// Käydään while loopissa tiedoston merkit läpi
while (fgets(uusi,2,tiedosto) != NULL) {
    // Jos merkit eroavat toisistaan ollaan saavuttu uuden merkin kohdalle.
    if (strstr(vanha,uusi) == NULL ) {
        if (b == 1) {
            ascii = (int)*vanha;
            fwrite(&luku,4,1,stdout);
            fwrite(&ascii,1,1,stdout);
            luku = 1;
        }
    // Jos merkit ovat samat lisätään lukua, joka kertoo, kuinka monta merkkiä on peräkkäin.
    } else {
        if (b == 1) {
            luku++;
        }
    }
    // Määritetään uusi merkki vanhaksi
    vanha = strdup(uusi);
    b = 1;
}
```

Kuva 4: *while*-looppi, jolla kirjoitetaan yhden tiedoston merkit *standard output*:iin.

Viimeisenä projektin toisessa vaiheessa tein $my - zip$-komennon, joka purkaa $my - unzip$-komennolla pakatun tiedoston. Tein *while*-loopin, joka käy iteraatioita läpi niin kauan, kunnes parametrina annetun tiedoston loppu on saavutettu. Luin aina loopissa jokaisella iteraatiolla ensin määrän, kuinka monta merkkiä on peräkkäin ja sitten itse merkin ASCII arvon. Tein sitten jokaisella iteraatiolla uuden *while*-loopin, jossa kirjoitin niin monta merkkiä peräkkäin standard input:iin kuin niiden määrä oli. Lopuksi tein koodin ympärille for-loopin, jossa voidaan käydä läpi niin monta tiedostoa kuin ohjelmalle kutsussa on annettu parametreina.

```
// Kirjoitetaan merkkejä niin kauan, kun askeltaja on pienempi, kuin merkkien määrä
i = 0;
while (i < luku) {
    printf("%c",kirjain);
    i++;
}
```

Kuva 5: *while*-looppi, jolla kirjoitetaan peräkkäiset samat merkit *standard output*:iin.

3 Osa 3: Unix Shell

Projektin viimeisessä osassa ideana on ohjelmoida yksinkertainen komentorivi tulkki *wish*, jolla voidaan ajaa muita komentoja. Ideana on, että kun *wish*-

ohjelman komentoriville syötetään jonkin komennon nimi, niin *wish*-tulkin muodostaa lapsiprosessin, jossa komento ajetaan. Tulkin on toimittava kahdella eri tavalla. Sitä kutsuessa voidaan antaa parametrina tiedosto (bash script), josta tulkki lukee ja suorittaa komennot. Vastaavasti tulkkia voidaan kutsua ilman lisäparametria, jolloin se kysyy loopissa aina uuden suoritettavan komennon *standard input*:ista, kunnes käyttäjä lopettaa ohjelman tulkin sisään rakennetulla *exit*-komennolla. Lopetuskomennon lisäksi tulkissa on kaksi muuta sisään rakennettua komentoa. Toinen niistä on *cd*-komento, jolla voidaan vaihtaa kansiota, jossa työskennellään (working folder). Toinen taas on *path*-komento, jolla voidaan vaihtaa polkuja, joista tulkki etsii sille annettujen komentojen tiedostoja ajaakseen ne.

Aloitin ohjelman koodaamalla ihan perus *while*-loopin, joka tulostaa jokaisella iteraatiolla tekstin *wish* > kertoakseen, että *wish*-tulkki on valmis seuraavaan komentoon. Tämän jälkeen ohjelmoin sisään rakennetun *exit*-komennon. Käytin tähän apuna *strstr* sekä *strlen* funktioita. Kun olin saanut *while*-loopin ja *exit* komennon toimimaan, perehdyin uuden lapsiprosessin luomiseen ja komennon sisältä uuden komennon ajamiseen. Opettelin käyttämään *fork*, *execv* ja *wait* komentoja tekemellä yksinkertaisen esimerkkiohjelman.

```
int main() {
        printf("Tää tulee alussa\n");
        char *args[]={"./k","Bella",NULL};

        int rc = fork();

        if (rc < 0) {
            fprintf(stderr,"Fork failed\n");
        } else if (rc == 0) {
            execv(args[0],args);
            exit(0);
        } else {
            wait(NULL);
        }
        printf("Tää tulee lopussa\n ");

    return 0;
}
```

Kuva 6: Tekemäni esimerkkiohjelma *fork*-, *execv*- ja *wait*-funktioiden käytöstä

```c
#include<stdio.h>
#include<unistd.h>

int main(int argc, char *argv[]) {
    if (argc == 3) {
        printf("%s ja %s sanoo WUH WUHWUHWUH!\n",argv[1],argv[2]);
    } else {
        printf("%s sanoo WUH!\n",argv[1]);
    }
    return 0;
}
```

Kuva 7: Tein yksinkertaisen pienen C-ohjelman, jota yritin ajaa *execv*-funktiota käyttäen.

Seuraavaksi ohjelmoin aliohjelman nimelta *makeArrey*, jonka avulla pilkoin *strtok* funktiota käyttäen tulkille syötetyn komennon argumentit välilyönnista ja rivinvaihtomerkista. Tallensin pilkotut argumentit ensin linkitettyyn listaan, jotta tietäisin, kuinka monta argumenttia on. Tämän jälkeen muodostin linkitetyn listan avulla taulukon (arrayn) argumenteista ja laitoin taulukon viimeiseksi alkioksi $NULL$:in. Näin sain muodostettua taulukon argumenteista, joka voidaan antaa *execv* funktiolle parametrina. Palautin aliohjelmasta tämän taulukon.

```c
/* Määritetään uusi askeltaja arrayn muodostamista varten. */
int index2 = 0;

/*Varataan muisti arraylle ja tallennetaan linkitetyn listan alkiot siihen.  */
pTemp = pStart;

char** arguments = (char**)malloc((index1+1)*sizeof(char*));

while(index2 < index1) {
    arguments[index2] = strdup(pTemp->pValue);
    index2++;
    pTemp = pTemp->pNext;
}

/* Määritetään arrayn viimeinen alkio nulliksi, (execv funktion parametrin vaatima muoto)*/
arguments[index2] = NULL;
```

Kuva 8: Linkitetyn listan muuttaminen *arguments*-taulukoksi

Tein komennon ajamista varten oman aliohjelmansa, jolle annoin nimeksi *runProgram*. Loin tämän aliohjelman sisällä uuden lapsiprosessin *fork*-funktiolla ja suoritin komennon *execv*-funktiolla. Laitoin alkuperäisen prosessin odottamaan lapsiprosessin valmistumista *wait*-funktiolla. Tässä vaiheessa kokosin ensimmäisen version tulkistani. Tulkkia pystyi nyt kutsumaan ainoastaan ilman parametreja. Komentoja pystyi lukemaan yksi kerrallaan *standard input*:sta *while*-loopissa. Komentojen lukeminen tiedostosta ei siis vielä toiminut. Komentoja ei pystynyt myöskään ajamaan, jos ne eivät löytyneet oletuspoluusta */bin*.

Kun olin saanut ensimmäisen version valmiiksi tein loput sisäänrakennetut komennot. Ensin tein *path* komennon (aliohjelman nimi *myPath*). Toteutin sen linkitetyn listan avulla. Sitten tein *cd* komennon (aliohjelman nimi *mycd*) *chdir* funktion avulla. Muodostin oman aliohjelman sitä varten, että tulkki tietää, onko kyseessä sisäänrakennettu komento (aliohjelman nimi *runningNavigation*). Tein myös oman aliohjelman, joka etsii tulkille syötettyä komentoa annetuista poluista (aliohjelman nimi *getRightPath*).

```
/* Käydään while-loopissa polut läpi ja tarkistetaan access-funktiolla, löytyykö etsitty komento jostain polusta */
while (pTemp != NULL) {

    /* Vartaan muisti apumuuttujaan. */
    currentPath = (char*)malloc(sizeof(pTemp->pValue)+sizeof(commandName)+sizeof(char)*2);

    /* Tarkistetaan, että muistin varaaminen onnistui. */
    if (currentPath == NULL) {
        char error_message[30] = "An error has occurred\n";
        write(STDERR_FILENO, error_message, strlen(error_message));
        return NULL;
    }

    /* Kirjoitetaan polku, johon on lisätty komennon nimi perään, muuttujaan currentPath. */
    sprintf(currentPath,"%s/%s",pTemp->pValue,commandName);
    if (access(currentPath,X_OK) == 0) {
        /* Jos polku löyty palautetaan se ja vapautetaan apumuuttujan viemä muisti. */
        free(currentPath);
        return pTemp->pValue;
    }
    /* Etsitään uusi alkio linkitetystä listasta ja tyhjennetään currentPath apumuuttujan varaama muisti.  */
    pTemp=pTemp->pNext;
    free(currentPath);
}
```

Kuva 9: Komennon etsiminen poluista *getRightPath*-aliohjelmassa *access*-funktiota hyödyntäen

Seuraavaksi keskityin siihen, että muisti varataan ja vapautetaan oikein. Selkeytin myös linkitettyjen listojen implemantaatioita luomalla erikseen aliohjelmat uuden linkitetyn listan alkion luomiselle (aliohjelman nimi *newLinkedListItem*) ja linkitetyn listan varaaman muistin vapauttamiselle (aliohjelman nimi *memoryRelease*). Mietin tässä kohtaa myös erilaisia virheskenaarioita. Ymmärsin, että *execv*-funktio lopettaa meneillään olevan prosessin ja vapauttaa aikaisemman prosessin muistit uuden prosessin käyttöön. Siis jos *execv*-funktion ajo onnistuu, niin $fork$ funktiolla aloitetun lapsiprosessin muistinvapautuksista ei tarvitse huolehtia. Mietin kuitenkin, että jos *execv*-funktion ajaminen epäonnistuu, niin $fork$ funktiolla luotu lapsiprosessi on ajettava hallitusti loppuun. Siis lapsiprosessin muisti on tällöin vapautettava. Laitoin tätä ongelmaa varten *runProgram* aliohjelman palautamaan arvon -1 tapauksessa, jossa *execv* funktio epäonnistuu, jolloin aliohjelmaa kutsuttaessa tiedetään, jos kyseessä on lapsiprosessi, joka pitää lopettaa hallitusti.

```
/* Käytetään ajamiseen runProgram-funktiota. Annetaan parametreiksi komennon argumentit, polku, josta komento löytyy
sekä tiedosto, johon mahdollisesti uudenlleenohjataan ohjelman output.*/
if (runProgram(arguments,currentPath,redirectionFile) == -1) {
    /* Jos ohjelmanajaminen epäonnistuu lopetetaan ohjelma hallitusti. */
    memoryRelease(pFirstPath);
    pFirstPath = NULL;
    pFirstPath = newLinkedListItem("exit",pFirstPath);
};
```

Kuva 10: Tapauksen, jossa *execv*-funktio epäonnistuu, käsittely

Kun olin saanut ensimmäisen version tulkista toimimaan, aloitin tekemään toiminnallisuutta, joka mahdollistaa komentojen lukemisen myös tiedostosta. Tein tätä varten oman aliohjelman nimelta *bashMode*. Aluksi kutsuin *bashMode* aliohjelmaa, jos tulkkia ajettaessa annettiin enemmän kuin yksi parametri. Tällöin tulkitsin koodissani kaikki parametrit omiksi tiedostoikseen. Kävin niiden komennot läpi yksi kerrallaan *while*-loopissa. Tajusin kuitenkin, että tämä oli tehtävänannon vastaista, joten muokkasin koodini niin, että tulkkia kutsuttaessa voidaan ottaa vain yksi tiedosto parametrina. Muuten tulee virheilmoitus. Jos parametrejä on yksi (eli vain *./wish*) aletaan heti kysymään komentoja standard-input:istä *while*-loopissa. Toteutin tulkin niin, että se ei lopeta luettuaan tiedoston komennot, ellei tiedoston komentojen mukana oli sisäänrakennettu lopetuskäsky *exit*. Muuten tiedoston luettuaan tulkki jatkaa ajamista kysymällä komentoja *standard input*:ista.

```
while (i<argc) {
    mycd(pnp);
    FILE* tiedosto = NULL;
    tiedosto = fopen(argv[i],"r");
    if (tiedosto == NULL) {
        printf("Voi ei");
        exit(1);
    }
    char *rivi = NULL;
    size_t len = 0;
    while(feof(tiedosto) == 0) {
        if (getline(&rivi,&len,tiedosto) > 1) {
            char** alkiot = alkioittenMuodostus(rivi," \n");
            polkuAlku = ajonOhjaus(alkiot,polkuAlku);
            free(alkiot);
        }
    }
    free(rivi);
    fclose(tiedosto);
    i++;
}
```

Kuva 11: Ensimmäinen versio *bashMode*-aliohjelmasta, jossa pystyi lukemaan komentoja monesta tiedostosta

Seuraavaksi toteutin uudelleenohjauksen. Tein siis tulkkiin mahdollisuuden uudelleenohjata komentojen tulosteet käyttäjän itse valitsemaan tiedostoon *standard output*:in sijaan. Toteutin tämän alkuun *runProgram*-aliohjelmassa katsomalla oliko komennot sisältävän taulukon toiseksi viimeinen alkio >-merkki. Tällöin tulkitsin taulukon viimeisen alkion tiedostoksi, johon tulosteet uudelleenohjataan. Tämä versio uudelleenohjauksesta ei kuitenkaan tukenut tapaa ajaa komentoa, jossa ei laiteta välilöyntiä komennon viimeisen parametrin, uudenlleenohjausmerkin ja tiedoston väliin. Tein siis uudelleen ohjauksen toisella tavalla, jossa ennen kuin olin vielä kertaakaan pilkkonut komentoa, pilkoin sen >-merkistä. Nyt otin itse komennon ensimmäisen kerran pilkotusta osasta ja tiedoston uudelleenohjausta varten toisen kerran pilkotusta osasta. Nyt välilyönneillä ei ollut väliä.

```
if (index > 1) {
    if (strstr(commandArguments[index-2],">") != NULL) {


        FILE* t = fopen(commandArguments[index-1],"w");
        fclose(t);

        int myFile = open(commandArguments[index-1], O_WRONLY | O_CREAT | O_TRUNC);

        char** reCommandArguments = (char**)malloc((index-1)*sizeof(char*));
        int index2 = 0;
        while (index2<index-2) {
            reCommandArguments[index2] = commandArguments[index2];
            index2++;
        }
        reCommandArguments[index2] = NULL;

        int rc = fork();
        if (rc < 0) {
            fprintf(stderr,"Fork failed\n");
        } else if (rc == 0) {
            dup2(myFile,STDOUT_FILENO);
            if (execv(extendedCurrentPath,reCommandArguments) == -1) {
                returnValue = -1;
            };
        } else {
            wait(NULL);
        }
        free(reCommandArguments);
        close(myFile);
```

Kuva 12: Ensimmäinen versio uudelleenohjaukseen, toteutettu *runProgram*-aliohjelman sisällä

```
aarre@DESKTOP-Q3LUG7N:~/kysteemi/harkkatyo$ ./wish
wish> cat Astronaut.txt>Astronaut2.txt
wish> cat Astronaut2.txt
Da-da-da-da, da-da-da-da, da
You and I talked about everything under the sun, and
```

Kuva 13: Ajamistyyli, jota ensimmäninen versio uudelleenohjauksesta ei tukenut

```
/* Pilkotaan komento ">"-merkistä mahdollista uudelleen ohjausta varten. */
char* redirectionCommand = strtok(commands[0],">");

/* Otetaan talteen tiedosto, johon output uudelleen ohjattaisiin*/
char* reHelp = strtok(NULL,">");

/* Jos argumentteja oli enemmän kuin yksi ">"-merkin jälkeen, niin annetaan virheilmoitus ja poistutaan aliohjelmasta */
char* redirectionFile = strtok(reHelp," >\n");
if (strtok(NULL," >\n") != NULL && redirectionFile != NULL) {
    char error_message[30] = "An error has occurred\n";
    write(STDERR_FILENO, error_message, strlen(error_message));
    free(commands);
    return pFirstPath;
}
```

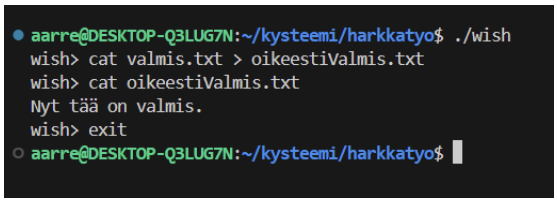
Kuva 14: Lopullinen versio uudelleenohjaukseen, toteutettu ennen *runProgram*-aliohjelman kutsumista

Viimeisenä tein tulkkiin mahdollisuuden ajaa komentoja rinnakkain. Tätä varten tein uuden aliohjelman *parallelHelp*. Muutin koko tulkkia niin, että kaikki komentojen ajaminen tapahtuu tämän aliohjelman kautta. Pilkoin tässä aliohjelmassa aina alkuun käyttäjän antaman komennon &-merkistä. Käytin pilkkomiseen jo aikaisemmin tekemääni *makeArrey*-aliohjelmaa. Tämän jälkeen tein *while*-loopin, joka tarkistaa *makeArrey*-aliohjelman tekemän taulukon pituuden. Jos taulukon pituus on vaan 1, niin komennon ajaminen suoritetaan normaalisti, niin kuin ennenkin. Jos taas taulukon pituus on enemmän, niin suoritetaan rinnakkaisajo. Rinnakkaisajon suoritin *while*-loopilla, joka tekee niin monta iteraatioita, kun &-merkistä pilkotussa taulukossa oli alkoita. Tein jokaiselle iteraatiolle $if-else$-rakenteen, jossa prosessejen tunnisteista katsottiin, onko kyseessä uusi lapsiprosessi vai alkuperäinen prosessi. Jos kyseessä on uusi lapsiporsessi, niin suoritetaan tämän iteraation indeksin kohdalla taulukossa oleva komento ja sen jälkeen lopetetaan prosessi. Jos taas kyseessä on alkuperäinen prosessi, niin mennään seuraavaan iteraatioon muodostamaan uusi prosessi, kunnes kaikki taulukon alkiot on käyty läpi. Jokaisella iteraatiolla keräsin alkuperäisessä prosessissa lapsiprosessien tunnisteen taulukkoon *pids*. Loopin päätyttyä alkuperäinen prosessi odottaa *waitpid*-funktion avulla kaikkien näiden lapsiprosessien valmistumista.

```
/* Laitetaan alkuperäinen prosessi odottamaan muita prosesseja tunnisteet sisältävän arreyn pids avulla*/
index = 0;
while(index < commandCounter) {
    waitpid(pids[index],NULL,0);
    index++;
}
```

Kuva 15: Lapsiprosessien valmistumisen odottaminen rinnakaisessa ajossa

Kun olin saanut ohjelman kaikki toiminnallisuudet tehtyä. Testailin ohjelmaa ja parantelin koodia. Otin turhia välivaiheita pois ja lisäsin unohtuneita kommennetteja ja virheenkäsittelyjä.



Kuva 16: Tulkin viimeinen testaus

4 Lähteet

Alla on lähteet, joita käytin ohjelmoinnin apuna.

- LUT:in C-ohjelmointiopas
- GeeksforGeeks nettisivu
- Stack Overflow nettisivu