Wydział informatyki Politechniki Białostockiej Przedmiot: Systemy operacyjne	Data: 21.04.2022
Projekt 1 – interfejs wywołań systemowych Linuksa. Temat 2 – Demon synchronizujący dwa podkatalogi. Grupa: PS5 1. Mateusz Maksimowicz 2. Klaudia Mieczkowska	Prowadzący: mgr inż. Tomasz Kuczyński

Temat 1 – Demon synchronizujący dwa podkatalogi.

[12p.] Program który otrzymuje co najmniej dwa argumenty: ścieżkę źródłową oraz ścieżkę docelową. Jeżeli któraś ze ścieżek nie jest katalogiem program powraca natychmiast z komunikatem błędu. W przeciwnym wypadku staje się demonem. Demon wykonuje następujące czynności: śpi przez pięć minut (czas spania można zmieniać przy pomocy dodatkowego opcjonalnego argumentu), po czym po obudzeniu się porównuje katalog źródłowy z katalogiem docelowym. Pozycje które nie są zwykłymi plikami są ignorowane (np. katalogi i dowiązania symboliczne). Jeżeli demon (a) napotka na nowy plik w katalogu źródłowym, i tego pliku brak w katalogu docelowym lub (b) plik w katalogu źródłowym ma późniejszą datę ostatniej modyfikacji demon wykonuje kopię pliku z katalogu źródłowego do katalogu docelowego - ustawiając w katalogu docelowym datę modyfikacji tak aby przy kolejnym obudzeniu nie trzeba było wykonać kopii (chyba że plik w katalogu źródłowym zostanie ponownie zmieniony). Jeżeli zaś odnajdzie plik w katalogu docelowym, którego nie ma w katalogu źródłowym to usuwa ten plik z katalogu docelowego. Możliwe jest również natychmiastowe obudzenie się demona poprzez wysłanie mu sygnału SIGUSR1. Wyczerpująca informacja o każdej akcji typu uśpienie/obudzenie się demona (naturalne lub w wyniku sygnału), wykonanie kopii lub usunięcie pliku jest przesłana do logu systemowego. Informacja ta powinna zawierać aktualną datę.

Dodatkowo:

A. [10p.] Dodatkowa opcja -R pozwalająca na rekurencyjną synchronizację katalogów (teraz pozycje będące katalogami nie są ignorowane). W szczególności jeżeli demon stwierdzi w katalogu docelowym podkatalog którego brak w katalogu źródłowym powinien usunąć go wraz z zawartością.
B. [12p.] W zależności od rozmiaru plików dla małych plików wykonywane jest kopiowanie przy pomocy read/write a w przypadku dużych przy pomocy mmap/write (plik źródłowy) zostaje zamapowany w całości w pamięci. Próg dzielący pliki małe od dużych może być przekazywany jako opcjonalny argument.

Program został podzielony na trzy pliki w zależności od ich funkcjonalności.

W głównym pliku **“main.c”** znajduje się funkcja main, główna funkcja demona, funkcja obsługująca otrzymywane sygnały oraz zmienna globalna oznaczającą tryb działania programu.

Zmienna trybu programu

```
enum
{
    DEFAULT,
    RECURSIVE
} mode = DEFAULT;
```

W funkcji main znajduje się obsługa argumentów podanych w terminalu.

Domyślne parametry to:

- czas uśpienia demona - 300 sekund,
- maksymalny rozmiar “malego” pliku - 1048576 bajty
- tryb działania programu - brak rekurencji

```
// Wczytanie argumentów
if (argc < 3)
{
    fprintf(stderr, "Usage: %s [-b size] [-s time] [-l size] [-R] source_directory destination_directory\n",
                argv[0]);
    exit(EXIT_FAILURE);
}


int opt, sleep_time = 300;

ssize_t buffor_size = 16384,
        large_file_size_limit = 1048576;

while ((opt = getopt(argc, argv, "Rb:s:l:")) != -1)
{
    switch (opt)
    {
    case 'R':
        mode = RECURSIVE;
        break;

    case 'b':
        buffor_size = atoi(optarg);
        if (buffor_size <= 0)
        {
            fprintf(stderr, "Expected argument after option -b\n");
            exit(EXIT_FAILURE);
        }
        break;
```

```
    case 's':
        sleep_time = atoi(optarg);
        if (sleep_time <= 0)
        {
            fprintf(stderr, "Expected argument after option -s\n");
            exit(EXIT_FAILURE);
        }
        break;

    case 'l':
        large_file_size_limit = atoi(optarg);
        if (large_file_size_limit <= 0)
        {
            fprintf(stderr, "Expected argument after option -l\n");
            exit(EXIT_FAILURE);
        }
        break;

    default:
        fprintf(stderr, "Usage: %s [-b size] [-s time] [-l size] [-R] source_directory destination_directory\n",
                argv[0]);
        exit(EXIT_FAILURE);
    }
}


if (is_regular_file(argv[optind], 0))
{
    fprintf(stderr, "%s nie jest folderem\n", argv[optind]);
    exit(EXIT_FAILURE);
}

if (is_regular_file(argv[optind + 1], 0))
{
    fprintf(stderr, "%s nie jest folderem\n", argv[optind + 1]);
    exit(EXIT_FAILURE);
}
```

Następnie tworzony jest demon i dodawana obsługa sygnału

```
pid_t pid = fork();

if (pid < 0)
    exit(EXIT_FAILURE);
else if (pid > 0)
    exit(EXIT_SUCCESS);

const int ssid = setsid();
if (ssid < 0)
    exit(EXIT_FAILURE);

pid = fork();

if (pid < 0)
    exit(EXIT_FAILURE);
else if (pid > 0)
    exit(EXIT_SUCCESS);

signal(SIGUSR1, handle_signal);
```

Następnie demon jest usypiany oraz uruchamiana jest główna funkcja demona - copy_and_delete_all_files

```
send_syslog(LOG_INFO, "Start demona. Uspanie na %d sekund", sleep_time);
sleep(sleep_time);

send_syslog(LOG_INFO, "%s", "Start kopiowania");
copy_and_delete_all_files(argv[optind], argv[optind + 1], buffor_size, large_file_size_limit);
send_syslog(LOG_INFO, "%s", "Skopiowane. Koniec demona.");

exit(EXIT_SUCCESS);
```

Funkcja obsługująca sygnał

```
void handle_signal(const int signum)
{
    if (signum == SIGUSR1)
        send_syslog(LOG_INFO, "%s", "SIGUSR1");
}
```

Główna funkcja demona

W nagłówku funkcji znajdują się ścieżki do folderów, rozmiar buforu używanego do kopiowania plików oraz rozmiar po którym plik jest traktowany jako “duży”.

```
void copy_and_delete_all_files(const char *source_path,
                               const char *destination_path,
                               const ssize_t buffor_size,
                               const ssize_t large_file_size_limit)
```

Na początku funkcji wczytywana jest lista plików z obu folderów oraz ich ilość.

```
// Lista plików w folderze źródłowym
struct dirent **source_files_list;

// Ilość plików w folderze źródłowym
const int no_of_source_files = scandir(source_path, &source_files_list, NULL, alphasort);

// Lista plików w folderze docelowym
struct dirent **dest_files_list;

// Ilość plików w folderze docelowym
const int no_of_dest_files = scandir(destination_path, &dest_files_list, NULL, alphasort);
```

Oraz tworzone zmienne to przetrzymywania tymczasowych ścieżek do plików.

```
char *src, *dst; // Ścieżki do plików
```

Następnie wykonywana jest pętla kopiująca pliki.

```
for (int i = 0; i < no_of_dest_files; i++)
{
    // Nazwa pliku źródłowego
    const char *source_file_name = source_files_list[i]->d_name;

    // Foldery "." ".." są omijane
    if (skip_location(source_file_name))
        continue;

    // W trybie domyślnym foldery są omijane
    if (mode == DEFAULT && is_directory(source_files_list[i]))
        continue;

    // Ścieżka do pliku źródłowego
    src = concat_path(source_path, source_file_name);
    // Ścieżka do pliku docelowego
    dst = concat_path(destination_path, source_file_name);
    // W trybie rekursywnym foldery są kopiowane
    if (mode == RECURSIVE && is_directory(source_files_list[i]))
    {
        DIR *dir = opendir(dst);
        if (!dir) // Sprawdza czy folder istnieje w katalogu docelowym
        {
            const int prem = get_permission(src);
            // Jeśli nie istnieje tworzy nowy folder o takiej samej nazwie i uprawnieniach
            if (mkdir(dst, prem) < 0)
            {
                fprintf(stderr, "copy_and_delete_all_files() mkdir() %s %s", dst, strerror(errno));
                exit(EXIT_FAILURE);
            }

            send_syslog(LOG_INFO, "Utworzono folder %s", dst);
        }

        free(dir);

        // Porównuje czasy modyfikacji folderów
        if (compare_files_times(src, dst))
        {
            // Uruchomienie rekurencyjne funkcji dla folderu
            copy_and_delete_all_files(src, dst, buffor_size, large_file_size_limit);
            // Zmiana daty modyfikacji po skopiowaniu na prawidłową
            copy_file_dates(src, dst);
        }

        free(dst);
        free(src);
        continue;
    }
```

```
    // Zmienne do zapisania informacji czy plik będzie kopiowany i
    // czy istnieje w folderze docelowym
    int copy = 1, was = 0;
    // Wszystkie pliki w folderze docelowym są sprawdzane
    for (int j = 0; j < no_of_dest_files; j++)
    {
        // Nazwa pliku docelowego
        const char *dest_file_name = dest_files_list[j]->d_name;

        // Foldery "." ".." są omijane
        if (skip_location(dest_file_name))
            continue;

        free(dst);
        // Ścieżka do pliku docelowego
        dst = concat_path(destination_path, dest_file_name);

        // Sprawdza czy pliki mają taką samą nazwę
        if (strcmp(source_file_name, dest_file_name) == 0)
        {
            // Ustala że plik był w folderze docelowym
            was = 1;
            // Porównuje czasy modyfikacji plików
            const long time = compare_files_times(src, dst);

            // Jeśli pliki są identyczne nie są kopiowane
            if (!time)
                copy = 0;

            // Po znalezieniu pliku z identyczną nazwą pętla przeszukująca
            // folder docelowy kończy się
            break;
        }
    }

    if (!copy)
    {
        free(src);
        free(dst);
        continue;
    }

    if (was) // Jeśli plik istnieje w folderze docelowym
    {
        // Istniejący plik jest usuwany
        delete_file(dst);
        // i kopiowany
        copy_file(src, dst, buffor_size, large_file_size_limit);
    }
    else // W przeciwnym wypadku jest tylko kopiowany
    {
        free(dst);
        dst = concat_path(destination_path, source_file_name);
        copy_file(src, dst, buffor_size, large_file_size_limit);
    }

    free(src);
    free(dst);
}
```

Następnie wykonywana jest pętla usuwająca pliki nieistniejące w folderze źródłowym.

```
for (int i = 0; i < no_of_dest_files; i++)
{
    // Nazwa pliku w folderze docelowym
    const char *dest_file_name = dest_files_list[i]->d_name;

    // Foldery "." ".." są omijane
    if (skip_location(dest_file_name))
        continue;

    // W trybie domyślnym foldery sa omijane
    if (mode == DEFAULT && is_directory(dest_files_list[i]))
        continue;

    // Zmienna zapisująca informację o tym czy plik będzie usuwany
    int delete = 1;

    // Wszystkie pliki w folderze źródłowym są sprawdzane
    for (int j = 0; j < no_of_source_files; j++)
    {
        // Nazwa pliku w folderze źródłowym
        const char *source_file_name = source_files_list[j]->d_name;

        // Foldery "." ".." są omijane
        if (skip_location(source_file_name))
            continue;

        // Jeśli plik o tej nazwie istnieje w folderze źródłowym
        // to plik nie jest usuwany
        if (strcmp(source_file_name, dest_file_name) == 0)
        {
            delete = 0;

            // Po znalezieniu pliku z identyczną nazwą pętla
            // przeszukująca folder źródłowy kończy się
            break;
        }
    }

    // Ścieżka do pliku
    dst = concat_path(destination_path, dest_file_name);
    if (delete)
    {
        // W zależności czy plik jest folderem czy zwykłym plikiem
        // odpowiednie funkcje są uruchamiane
        if (!is_regular_file(dst, 1))
            delete_directory(dst);
        else
            delete_file(dst);
    }

    free(dst);
}
```

Na koniec funkcji zwalniana jest pamięć.

```
for (int i = 0; i < no_of_source_files; i++)
    free(source_files_list[i]);

for (int i = 0; i < no_of_dest_files; i++)
    free(dest_files_list[i]);

free(dest_files_list);
free(source_files_list);
```

W pliku **files.c** znajdują się funkcje wykonujące operacje na plikach.

Nagłówki funkcji w pliku files.h

```
void delete_file(const char* path);
void delete_directory(const char* path);
void copy_file_dates(const char* from, const char* to);
void copy_file(const char* from, const char* to,
               ssize_t buffor, ssize_t large_file_size_limit);
```

Funkcja kopiująca plik.

W nagłówku funkcji znajdują się ścieżki do folderów, rozmiar buforu używanego do kopiowania plików oraz rozmiar po którym plik jest traktowany jako duży.

```
void copy_file(const char* from, const char* to,
               ssize_t buffor, ssize_t large_file_size_limit)
```

Na początku funkcji oba pliki są otwierane.

```
const int src = open(from, O_RDONLY);
if (src < 0)
{
    fprintf(stderr, "copy_file() open(from) %s %s", from, strerror(errno));
    exit(EXIT_FAILURE);
}

const int prem = get_permission(from);
const int dst = open(to, O_WRONLY | O_CREAT | O_APPEND, prem);
if (dst < 0)
{
    fprintf(stderr, "copy_file() open(to) %s %s", to, strerror(errno));
    exit(EXIT_FAILURE);
}
```

Następnie pobierany jest rozmiar pliku. W zależności od rozmiaru używana jest metoda read/write lub mmap.

```
struct stat st;
stat(from, &st);
const ssize_t size = st.st_size;

send_syslog(LOG_INFO, "Proba skopiowania z %s do %s", from, to);

if (size > large_file_size_limit) // Kopiowanie dużego pliku za pomocą munmap
{
    char *addr = mmap(NULL, size, PROT_READ, MAP_PRIVATE, src, 0);
    if (addr == MAP_FAILED)
    {
        fprintf(stderr, "copy_file() mmap() %s", from);
        exit(EXIT_FAILURE);
    }

    write(dst, addr, size);
    munmap(addr, size);
}
else // Kopiowanie za pomocą write
{
    void *buf = malloc(buffor);
    ssize_t bytes_read = read(src, buf, buffor);

    do
    {
        write(dst, buf, bytes_read);
    } while ((bytes_read = read(src, buf, buffor)) != 0);

    free(buf);
}

send_syslog(LOG_INFO, "Skopiowano z %s do %s", from, to);
```

Na końcu funkcji oba pliki są zamykane oraz data modyfikacji stworzonego pliku jest zmieniana na poprawną.

```
int err = close(src);
if (err < 0)
{
    fprintf(stderr, "copy_file() close(src) %s", strerror(errno));
    exit(EXIT_FAILURE);
}
err = close(dst);
if (err < 0)
{
    fprintf(stderr, "copy_file() close(dst) %s", strerror(errno));
    exit(EXIT_FAILURE);
}

// Zmiana daty modyfikacji pliku
copy_file_dates(from, to);
```

Funkcja edytująca datę modyfikacji.

```
void copy_file_dates(const char *from, const char *to)
{
    struct stat st;
    stat(from, &st);
    const long mtime = st.st_mtime;
    struct utimbuf ubuf;
    time(&ubuf.actime);
    ubuf.modtime = mtime;
    const int err = utime(to, &ubuf);
    if (err < 0)
    {
        fprintf(stderr, "copy_file_dates() utime() %s %s %s", from, to, strerror(errno));
        exit(EXIT_FAILURE);
    }
}
```

Funkcja usuwająca folder.

```
void delete_directory(const char *path)
{
    // Lista plików w folderze
    struct dirent **files_list;
    // Liczba plików w folderze
    const int no_of_files = scandir(path, &files_list, NULL, alphasort);
    char *src;
    send_syslog(LOG_INFO, "Proba skasowania folderu %s", path);

    // Wszystkie pliki w folderze są usuwane
    for (int i = 0; i < no_of_files; i++)
    {
        // Nazwa pliku
        const char *file_name = files_list[i]->d_name;
        // Foldery "." ".." są omijane
        if (skip_location(file_name))
            continue;

        src = concat_path(path, file_name);

        // Rekurencyjne usuwanie folderów
        if (is_directory(files_list[i]))
            delete_directory(src);
        else
            delete_file(src);

        delete_file(src);
        free(src);
    }

    // Usunięcie folderu po skasowaniu plików w znajdujących się w nim
    const int err = unlinkat(NULL, path, AT_REMOVEDIR);
    if (err < 0)
    {
        fprintf(stderr, "delete_file() %s %s", path, strerror(errno));
        exit(EXIT_FAILURE);
    }
    send_syslog(LOG_INFO, "Skasowano folder %s", path);

    // Zwalnianie pamięci
    for (int i = 0; i < no_of_files; i++)
```

```
        free(files_list[i]);

    free(files_list);
}
```

Funkcja usuwająca plik.

```
void delete_file(const char *path)
{
    send_syslog(LOG_INFO, "Proba skasowania %s", path);
    const int err = unlink(path);

    if (err < 0)
    {
        fprintf(stderr, "delete_file() %s %s", path, strerror(errno));
        exit(EXIT_FAILURE);
    }
    send_syslog(LOG_INFO, "Skasowano %s", path);
}
```

W pliku **utils.c** znajdują się funkcje.

Nagłówki funkcji w pliku utils.h

```
int is_directory(const struct dirent* dir);
int is_regular_file(const char* path, int error);
int get_permission(const char* path);
char* concat_path(const char* source, const char* file);
int skip_location(const char* name);
long compare_files_times(const char* file1, const char* file2);
void send_syslog(int type, const char* format, ...);
```

Funkcja łącząca podane łańcuchy znaków

```
char* concat_path(const char* source, const char* file)
{
    char* path = malloc(strlen(source) + strlen(file) + 2);

    strcpy(path, source);

    strcat(path, "/");
    strcat(path, file);

    return path;
}
```

Funkcja ta używa malloc, dlatego ważne jest użycie funkcji free po użyciu tej funkcji. W przeciwnym wypadku powstają wycieki pamięci.

Funkcja zwracająca uprawnienia pliku

```
int get_permission(const char* path)
{
    struct stat st;
    stat(path, &st);
    mode_t perm = st.st_mode;

    return (perm & S_IRUSR) | (perm & S_IWUSR) | (perm & S_IXUSR)
         | (perm & S_IRGRP) | (perm & S_IWGRP) | (perm & S_IXGRP)
         | (perm & S_IROTH) | (perm & S_IWOTH) | (perm & S_IXOTH);
}
```

Funkcja porównująca czas modyfikacji pliku

```
long compare_files_times(const char* file1, const char* file2)
{
    struct stat stat1;
    if (stat(file1, &stat1) == -1)
    {
        fprintf(stderr, "compare_files_times() %s %s", file1, strerror(errno));
        exit(EXIT_FAILURE);
    }

    struct stat stat2;
    if (stat(file2, &stat2) == -1)
    {
        fprintf(stderr, "compare_files_times() %s %s", file2, strerror(errno));
        exit(EXIT_FAILURE);
    }

    const long time1 = stat1.st_mtime;
    const long time2 = stat2.st_mtime;

    return time1 - time2;
}
```

Funkcja wysyłająca informację do dziennika systemowego wraz z aktualną datą i godziną

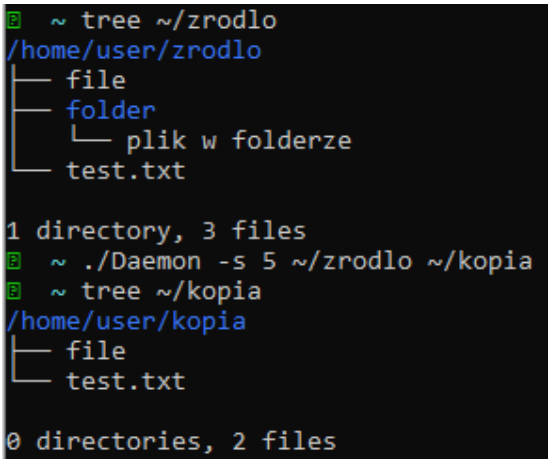
```
void send_syslog(const int type, const char* format, ...)
{
    char buffer[1024];
    va_list args;
    va_start(args, format);
    vsprintf(buffer, format, args);
    va_end(args);

    time_t rawtime;
    time(&rawtime);
    const struct tm* ti = localtime(&rawtime);

    syslog(type, "%d.%d.%d-%d:%d:%d %s",
           ti->tm_mday, ti->tm_mon + 1, ti->tm_year + 1900,
           ti->tm_hour, ti->tm_min, ti->tm_sec, buffer);
}
```

Przykłady działania demona

1. Działanie demona bez rekurencji

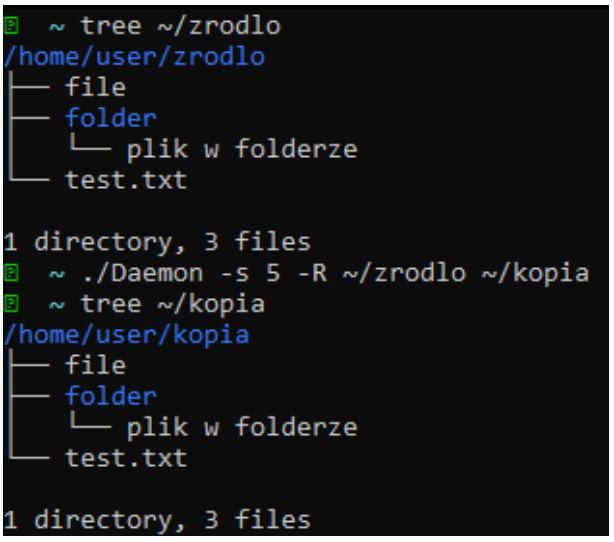


Rys. 1.1 Struktura katalogu źródłowego i docelowego po działaniu programu

```
Daemon[7629]: 25.4.2022-18:35:47 Start demona. Uspanie na 5 sekund
Daemon[7629]: 25.4.2022-18:35:52 Start kopiowania
Daemon[7629]: 25.4.2022-18:35:52 Proba skopiowania z /home/user/zrodlo//file do /home/user/kopia//file
Daemon[7629]: 25.4.2022-18:35:52 Skopiowano z /home/user/zrodlo//file do /home/user/kopia//file
Daemon[7629]: 25.4.2022-18:35:52 Proba skopiowania z /home/user/zrodlo//test.txt do /home/user/kopia//test.txt
Daemon[7629]: 25.4.2022-18:35:52 Skopiowano z /home/user/zrodlo//test.txt do /home/user/kopia//test.txt
Daemon[7629]: 25.4.2022-18:35:52 Skopiowane.
Daemon[7628]: 25.4.2022-18:35:52 Koniec demona.
```

Rys. 1.2 Zapisy z dziennika systemowego

2. Działanie demona z rekurencją



Rys. 2.1 Struktura katalogu źródłowego i docelowego po działaniu programu

```
Daemon[1800]: 25.4.2022-22:17:7 Start demona. Uspanie na 5 sekund
Daemon[1800]: 25.4.2022-22:17:12 Start kopiowania
Daemon[1800]: 25.4.2022-22:17:12 Proba skopiowania z /home/user/zrodlo//file do /home/user/kopia//file
Daemon[1800]: 25.4.2022-22:17:12 Skopiowano z /home/user/zrodlo//file do /home/user/kopia//file
Daemon[1800]: 25.4.2022-22:17:12 Utworzono folder /home/user/kopia//folder
Daemon[1800]: 25.4.2022-22:17:12 Proba skopiowania z /home/user/zrodlo//folder/plik w folderze do /home/user/kopia//folder/plik w folderze
Daemon[1800]: 25.4.2022-22:17:12 Skopiowano z /home/user/zrodlo//folder/plik w folderze do /home/user/kopia//folder/plik w folderze
Daemon[1800]: 25.4.2022-22:17:12 Proba skopiowania z /home/user/zrodlo//test.txt do /home/user/kopia//test.txt
Daemon[1800]: 25.4.2022-22:17:12 Skopiowano z /home/user/zrodlo//test.txt do /home/user/kopia//test.txt
Daemon[1800]: 25.4.2022-22:17:12 Skopiowane. Koniec demona.
```

Rys. 2.2 Zapisy z dziennika systemowego

Demon po odnalezieniu pliku z tą samą datą modyfikacji nie kopiuje go

```
Daemon[1895]: 25.4.2022-22:19:44 Start demona. Uspanie na 5 sekund
Daemon[1895]: 25.4.2022-22:19:49 Start kopiowania
Daemon[1895]: 25.4.2022-22:19:49 Skopiowane. Koniec demona.
```

Rys. 2.3 Zapisy z dziennika systemowego po ponownym uruchomieniu demona

Jeśli zmienimy datę modyfikacji pliku zostanie on skopiowany

```
Daemon[1969]: 25.4.2022-22:22:5 Start demona. Uspanie na 5 sekund
Daemon[1969]: 25.4.2022-22:22:10 Start kopiowania
Daemon[1969]: 25.4.2022-22:22:10 Proba skasowania /home/user/kopia/test.txt
Daemon[1969]: 25.4.2022-22:22:10 Skasowano /home/user/kopia/test.txt
Daemon[1969]: 25.4.2022-22:22:10 Proba skopiowania z /home/user/zrodlo/test.txt do /home/user/kopia/test.txt
Daemon[1969]: 25.4.2022-22:22:10 Skopiowano z /home/user/zrodlo/test.txt do /home/user/kopia/test.txt
Daemon[1969]: 25.4.2022-22:22:10 Skopiowane. Koniec demona.
```

Rys. 2.4 Zapisy z dziennika systemowego po ponownym uruchomieniu demona

3. Kasowanie plików nieistniejących w folderze docelowym

Z katalogu źródłowego z podfolderu “folder” został usunięty plik “plik w folderze”. Znajduje się on w katalogu docelowym. Więc po uruchomieniu demona plik powinien zostać usunięty z folderu docelowego.

```
~ rm ~/zrodlo/folder/plik\ w\ folderze
~ tree ~/zrodlo
/home/user/zrodlo
├── file
├── folder
└── test.txt

1 directory, 2 files
~ ./Daemon -s 5 -R ~/zrodlo ~/kopia
~ tree ~/kopia
/home/user/kopia
├── file
├── folder
│   └── plik w folderze
└── test.txt

1 directory, 3 files
~ tree ~/kopia
/home/user/kopia
├── file
├── folder
└── test.txt

1 directory, 2 files
```

Rys. 3.1 Struktura katalogu źródłowego i docelowego po działaniu programu

```
Daemon[11662]: 25.4.2022-18:48:38 Start demona. Uspanie na 5 sekund
Daemon[11662]: 25.4.2022-18:48:43 Start kopiowania
Daemon[11662]: 25.4.2022-18:48:43 Proba skasowania /home/user/kopia/folder/plik w folderze
Daemon[11662]: 25.4.2022-18:48:43 Skasowano /home/user/kopia/folder/plik w folderze
Daemon[11662]: 25.4.2022-18:48:43 Skopiowane. Koniec demona.
```

Rys. 3.2 Zapisy z dziennika systemowego

4. Obsługa sygnału

```
~ tree ~/zrodlo
/home/user/zrodlo
├── file
├── folder
│   └── plik w folderze
└── test.txt

1 directory, 3 files
~ ./Daemon -s 6000 -R ~/zrodlo ~/kopia
~ ps -e | grep Daemon
 11705 ?        00:00:00 Daemon
~ kill -SIGUSR1 11705
~ tree ~/kopia
/home/user/kopia
├── file
├── folder
│   └── plik w folderze
└── test.txt

1 directory, 3 files
```

Rys. 4.1 Struktura katalogu źródłowego i docelowego oraz wykonanie polecenia kill.

```
Daemon[11705]: 25.4.2022-18:51:36 Start demona. Uspanie na 6000 sekund
Daemon[11705]: 25.4.2022-18:52:34 SIGUSR1
Daemon[11705]: 25.4.2022-18:52:34 Start kopiowania
Daemon[11705]: 25.4.2022-18:52:34 Proba skopiowania z /home/user/zrodlo/file do /home/user/kopia/file
Daemon[11705]: 25.4.2022-18:52:34 Skopiowano z /home/user/zrodlo/file do /home/user/kopia/file
Daemon[11705]: 25.4.2022-18:52:34 Utworzono folder /home/user/kopia/folder
Daemon[11705]: 25.4.2022-18:52:34 Proba skopiowania z /home/user/zrodlo/folder/plik w folderze do /home/user/kopia/folder/plik w folderze
Daemon[11705]: 25.4.2022-18:52:34 Skopiowano z /home/user/zrodlo/folder/plik w folderze do /home/user/kopia/folder/plik w folderze
Daemon[11705]: 25.4.2022-18:52:34 Proba skopiowania z /home/user/zrodlo/test.txt do /home/user/kopia/test.txt
Daemon[11705]: 25.4.2022-18:52:34 Skopiowano z /home/user/zrodlo/test.txt do /home/user/kopia/test.txt
Daemon[11705]: 25.4.2022-18:52:34 Skopiowane. Koniec demona.
```

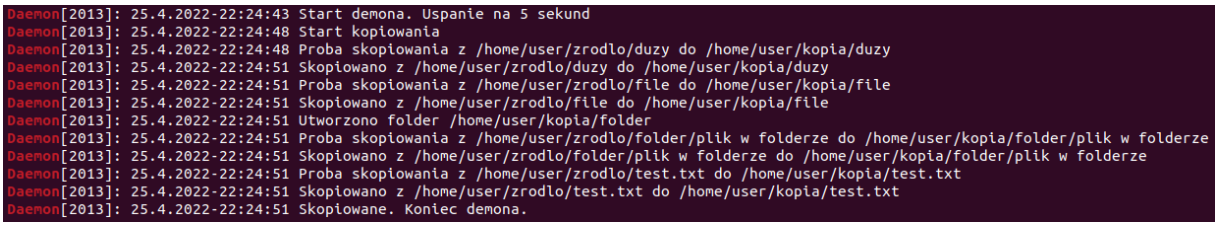
Rys. 4.2 Zapisy z dziennika systemowego

5. Kopiowanie dużych plików

Przy uruchomieniu demona został użyty parametr -b pozwalający ustalić maksymalny rozmiar “małego” pliku. Po przekroczeniu tego rozmiaru pliki są kopiowane przy pomocy mmap.

```
~ ls -l ~/zrodlo
total 1048588
-rw-rw-r-- 1 user user 1073741824 kwi 25 22:23 duzy
-rw-rw-r-- 1 user user          0 kwi 25 18:28 file
drwxrwxr-x 2 user user       4096 kwi 25 22:16 folder
-rw-rw-r-- 1 user user          2 kwi 25 22:18 test.txt
~ ./Daemon -s 5 -b 1 -R ~/zrodlo ~/kopia
~ ls -l ~/kopia
total 1048588
-rw-rw-r-- 1 user user 1073741824 kwi 25 22:23 duzy
-rw-rw-r-- 1 user user          0 kwi 25 18:28 file
drwxrwxr-x 2 user user       4096 kwi 25 22:16 folder
-rw-rw-r-- 1 user user          2 kwi 25 22:18 test.txt
```

Rys. 5.1 Struktura katalogu źródłowego i docelowego po działaniu programu



Rys. 5.2 Zapisy z dziennika systemowego

6. Sprawdzenie wycieków pamięci i błędów za pomocą programu Valgrind

Program został przetestowany za pomocą programu Valgrind. Aby wykonać test z funkcji main została usunięta funkcja fork, aby nie tworzyć oddzielnych procesów, których valgrind nie mógłby sprawdzić. Program został przetestowany dla rozbudowanego folderu źródłowego.

Program był uruchamiany cztery razy, przy każdym uruchomieniu zachodzą poniższe warunki:

1. Katalog docelowy jest pusty
2. W katalogu docelowym istnieją pliki nieistniejące w katalogu źródłowym oraz pliki z nieprawidłową datą modyfikacji
3. Katalog docelowy i źródłowy jest identyczny
4. Katalog źródłowy jest pusty a w katalogu docelowym znajdują się pliki skopiowane wcześniej

```
~ valgrind ./Daemon -R -s 1 -b 1024 ~/zrodlo ~/kopia
==16789== Memcheck, a memory error detector
==16789== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==16789== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==16789== Command: ./Daemon -R -s 1 -b 1024 /home/user/zrodlo /home/user/kopia
==16789==
==16789==
==16789== HEAP SUMMARY:
==16789==     in use at exit: 0 bytes in 0 blocks
==16789==   total heap usage: 12,518 allocs, 12,518 frees, 37,785,269 bytes allocated
==16789==
==16789== All heap blocks were freed -- no leaks are possible
==16789==
==16789== For lists of detected and suppressed errors, rerun with: -s
==16789== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
~ valgrind ./Daemon -R -s 1 -b 1024 ~/zrodlo ~/kopia
==20785== Memcheck, a memory error detector
==20785== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==20785== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==20785== Command: ./Daemon -R -s 1 -b 1024 /home/user/zrodlo /home/user/kopia
==20785==
==20785==
==20785== HEAP SUMMARY:
==20785==     in use at exit: 0 bytes in 0 blocks
==20785==   total heap usage: 4,740 allocs, 4,740 frees, 12,189,779 bytes allocated
==20785==
==20785== All heap blocks were freed -- no leaks are possible
==20785==
==20785== For lists of detected and suppressed errors, rerun with: -s
==20785== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
~ valgrind ./Daemon -R -s 1 -b 1024 ~/zrodlo ~/kopia
==20788== Memcheck, a memory error detector
==20788== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==20788== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==20788== Command: ./Daemon -R -s 1 -b 1024 /home/user/zrodlo /home/user/kopia
==20788==
==20788==
==20788== HEAP SUMMARY:
==20788==     in use at exit: 0 bytes in 0 blocks
==20788==   total heap usage: 313 allocs, 313 frees, 273,387 bytes allocated
==20788==
==20788== All heap blocks were freed -- no leaks are possible
==20788==
==20788== For lists of detected and suppressed errors, rerun with: -s
==20788== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
~ rm -rf ~/zrodlo/*
zsh: sure you want to delete all 21 files in /home/user/zrodlo [yn]? y
~ valgrind ./Daemon -R -s 1 -b 1024 ~/zrodlo ~/kopia
==20799== Memcheck, a memory error detector
==20799== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==20799== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==20799== Command: ./Daemon -R -s 1 -b 1024 /home/user/zrodlo /home/user/kopia
==20799==
==20799==
==20799== HEAP SUMMARY:
==20799==     in use at exit: 0 bytes in 0 blocks
==20799==   total heap usage: 10,710 allocs, 10,710 frees, 28,698,486 bytes allocated
==20799==
==20799== All heap blocks were freed -- no leaks are possible
==20799==
==20799== For lists of detected and suppressed errors, rerun with: -s
==20799== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Rys. 6 Wynik działania programu Valgrind

Podział pracy w Projekcie 1:

Mateusz Maksimowicz - Program "Deamon", Sprawozdanie.
Klaudia Mieczkowska - Sprawozdanie.