Zespół:
Rafał Dąbrowski
Urszula Tokajuk

Grupa:
PS6

Prowadzący:
Dr. Inż. Wojciech Kwedlo

Temat 2 - Demon synchronizujący dwa podkatalogi

[12p] Program, który otrzymuje co najmniej dwa argumenty:

- Ścieżkę źródłową
- 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 zmienić przy pomocy dodatkowego opcjonalnego argumentu).
- Po obudzeniu się porównuje katalog źródłowy z katalogiem docelowym.
- Pozycje które nie są zwykłymi plikami są ignorowane (np. katalogi i dowiązania symboliczne).

Jeżeli demon:

- Napotka na nowy plik w katalogu źródłowym i tego pliku brak w katalogu docelowym.
- Plik w katalogu źródłowym ma późniejszą datę ostatniej modyfikacji.

To 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.

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.

Uwagi:

a) Wszelkie operacje na plikach należy wykonywać przy pomocy API Linuksa, a nie standardowej biblioteki języka C.
b) Kopiowanie za każdym obudzeniem całego drzewa katalogów zostanie potraktowane jako poważny błąd.
c) Podobnie jak przerzucenie części zadań na shell systemowy (funkcja system).

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <dirent.h>
#include <errno.h>
#include <signal.h>
#include <stdbool.h>
#include <string.h>
#include <limits.h>
#include <utime.h>
#include <time.h>
#include <syslog.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <linux/fs.h>

#define BUFFER_SIZE 131072
#define MAX_SLEEP_TIME 86400

size_t fileSizeThreshold = 1048576;
int sleepInterval = 300;
bool recursiveSearch = false;

/*LISTA*/
typedef struct node {     //     Struktura elementu listy
       char *filename;
       struct node *next;
} Node;

typedef struct list {     //     Struktura listy
       Node *head;
} List;

/*Funkcja tworząca nowy element listy*/
Node *CreateNode(char *filename) {
       Node *newNode = malloc(sizeof(Node));
       newNode->filename = filename;
       newNode->next = NULL;

       return newNode;
}

/*Funkcja inicjalizująca listę*/
List *InitList() {
       List *list = malloc(sizeof(List));
       list->head = NULL;

       return list;
}

/*Funkcja dołączająca element filename na koniec listy list*/
void Append(char *filename, List *list) {
       Node *current = NULL;
       if (list->head == NULL) {
              list->head = CreateNode(filename);
       }
       else {
              current = list->head;
              while (current->next != NULL) {
                     current = current->next;
              }

              current->next = CreateNode(filename);
       }
}
```

```
/*Funkcja, która zwraca informację, czy lista list zawiera plik o nazwie name*/
int Contains(List *list, char *name) {
	Node *current = list->head;
	while (current != NULL) {
		if (strcmp(current->filename, name) == 0) {
			return 1;
		}

		current = current->next;
	}

	return -1;
}

/*Funkcja niszczy liste list i dealokuje pamięć*/
void DestroyList(List *list) {
	Node *current;

	while ((current = list->head) != NULL) {
		list->head = list->head->next;
		free(current);
	}

	free(list);
}

/*Funkcja usuwa element node z listy list*/
void RemoveAt(Node *node, List *list) {
	Node *current = node;
	Node *previous = current;

	previous->next = current->next;
	if (current == list->head) {
		list->head = current->next;
	}

	free(current);
	return;
}
/*LISTA*/

/*Funkcja, która pobiera informacje o pliku path i zwraca wskaźnik na struct stat*/
struct stat *GetFileInfo(const char *path) {
	struct stat *fileInfo = malloc(sizeof(struct stat));

	if (stat(path, fileInfo) == -1) {
		syslog(LOG_ERR, "stat(): \"%s\" (%s)", path, strerror(errno));
		return NULL;
	}

	return fileInfo;
}

/*Funkcja, która ustawia czas modyfikacji do pliku destPath zgodnie z fileInfo i czas dostępu na
aktualny czas*/
int SyncModTime(struct stat *fileInfo, const char *destPath) {
	struct utimbuf newTime;

	newTime.actime = time(NULL);    //  Ustaw datę doestępu na datę teraźniejszą
	newTime.modtime = fileInfo->st_mtime;   //  Ustaw datę modyfiakcji na datę modyfikacji pliku
źródłowego
	if (utime(destPath, &newTime) == -1) {
		syslog(LOG_ERR, "utime(): \"%s\" (%s)", destPath, strerror(errno));
		return -1;
	}

	return 0;
}
```

```
/*Funkcja kopiująca plik srcPath do destPath, używając odwzorowania w pamięci*/
int MmapCopy(const char *srcPath, const char *destPath) {
	struct stat fileInfo;

	int source = open(srcPath, O_RDONLY);
	if (source == -1) {
		syslog(LOG_ERR, "open(): \"%s\" (%s)", srcPath, strerror(errno));
		return -1;
	}

	off_t fileSize;
	if (fstat(source, &fileInfo) == -1) {	//  Pobierz informację o pliku źródłowym
		syslog(LOG_ERR, "fstat(): \"%s\" (%s)", srcPath, strerror(errno));
		return -1;
	}

	fileSize = fileInfo.st_size;	//	Pobierz rozmiar pliku źródłowego

	int destination = open(destPath, O_RDWR | O_CREAT, fileInfo.st_mode);
	if (destination == -1) {
		syslog(LOG_ERR, "open(): \"%s\" (%s)", destPath, strerror(errno));
		return -1;
	}

	int *srcAddress = mmap(NULL, fileSize, PROT_READ, MAP_PRIVATE, source, 0);	//	Stwórz
odwzorowanie pliku źródłowego w pamięci
	if (srcAddress == MAP_FAILED) {
		syslog(LOG_ERR, "mmap(): \"%s\" (%s)", srcPath, strerror(errno));
		return -1;
	}

	if (ftruncate(destination, fileSize) == -1) {  //  Zmień rozmiar pliku docelowego na rozmiaru
pliku źródłowego
		syslog(LOG_ERR, "ftruncate(): \"%s\" (%s)", destPath, strerror(errno));
		return -1;
	}

	int *destAddress = mmap(NULL, fileSize, PROT_READ | PROT_WRITE, MAP_SHARED, destination,
0);	//  Stwórz odwzorowanie pliku docelowego w pamięci
	if (destAddress == MAP_FAILED) {
		syslog(LOG_ERR, "mmap(): \"%s\" (%s)", destPath, strerror(errno));
		return -1;
	}

	if (memcpy(destAddress, srcAddress, fileSize) == NULL) { //  Skopiuj zawartość odwzorowania
pliku źródłowego do miejsca odwzorowania pliku docelowego
		syslog(LOG_ERR,	"memcpy():	\"%s\"	to	\"%s\"	(%s)",	srcPath,	destPath,
strerror(errno));
		return -1;
	}

	if (munmap(srcAddress, fileSize) == -1) {	//  Usuń odwzorowanie pliku źródłowego
		syslog(LOG_ERR, "munmap(): \"%s\" (%s)", srcPath, strerror(errno));
		return -1;
	}

	if (munmap(destAddress, fileSize) == -1) {	//  Usuń odwzorowanie pliku docelowego
		syslog(LOG_ERR, "munmap(): \"%s\" (%s)", destPath, strerror(errno));
		return -1;
	}

	if (SyncModTime(&fileInfo, destPath) == -1) {	//  Ustaw datę modyfikacji pliku docelowego
na datę modyfikacji pliku źródłowego
		syslog(LOG_ERR, "SyncModTime(): \"%s\" Could not synchronize modification time.",
destPath);
		return -1;
	}
```

```
/*Zamknij deskryptory*/
	if (close(source) == -1) {
		syslog(LOG_ERR, "close(): \"%s\" (%s)", srcPath, strerror(errno));
		return -1;
	}

	if (close(destination) == -1) {
		syslog(LOG_ERR, "close(): \"%s\" (%s)", destPath, strerror(errno));
		return -1;
	}

	return 0;
}

/*Funkcja kopiująca plik srcPath do destPath, używając API Linuxa*/
int RegularCopy(const char *srcPath, const char *destPath) {
	char buffer[BUFFER_SIZE];
	struct stat fileInfo;

	int source = open(srcPath, O_RDONLY);
	if (source == -1) {
		syslog(LOG_ERR, "open(): \"%s\" (%s)", srcPath, strerror(errno));
		return -1;
	}

	if (fstat(source, &fileInfo) == -1) {
		syslog(LOG_ERR, "fstat(): \"%s\" (%s)", srcPath, strerror(errno));
		return -1;
	}

	int destination = open(destPath, O_WRONLY | O_CREAT | O_TRUNC, fileInfo.st_mode);
	if (destination == -1) {
		syslog(LOG_ERR, "open(): \"%s\" (%s)", destPath, strerror(errno));
		return -1;
	}

	ssize_t bytesRead;
	ssize_t bytesWritten;
	while ((bytesRead = read(source, &buffer, BUFFER_SIZE)) != 0) { //  Dopóki pobrano jakieś
dane
		if (bytesRead == -1) {
			if (errno == EINTR) {   //  Jeżeli odczyt został przerwany sygnałem
				continue;   //  Ponów próbę odczytu
			}

			syslog(LOG_ERR, "read(): \"%s\" (%s)", srcPath, strerror(errno));
			return -1;
		}

		bytesWritten = write(destination, &buffer, bytesRead);    //    Zapisz bytesRead
odczytanych bajtów
		if (bytesWritten == -1) {
			syslog(LOG_ERR, "write(): \"%s\" (%s)", destPath, strerror(errno));
			return -1;
		}

		if (bytesRead != bytesWritten) {    //  Jeżeli zapisano inną ilość bajtów niż odczytano
			syslog(LOG_ERR, "write(): \"%s\" (%s)", destPath, strerror(errno));
			return -1;
		}
	}

	if (SyncModTime(&fileInfo, destPath) == -1) {    // Ustaw datę modyfikacji pliku docelowego
na datę modyfikacji pliku źródłowego
		syslog(LOG_ERR, "SyncModTime(): \"%s\" Could not synchronize modification time.",
destPath);
		return -1;
	}
```

```
/*Zamknij deskryptory*/
	if (close(source) == -1) {
		syslog(LOG_ERR, "close(): \"%s\" (%s)", srcPath, strerror(errno));
		return -1;
	}

	if (close(destination) == -1) {
		syslog(LOG_ERR, "close(): \"%s\" (%s)", destPath, strerror(errno));
		return -1;
	}

	return 0;
}

/*Funkcja kopiująca plik srcPath do destPath, na podstawie rozmiaru pliku decyduje w jaki sposób
plik zostanie skopiowany*/
int Copy(const char *srcPath, const char *destPath) {
	struct stat *srcFileInfo = GetFileInfo(srcPath);

	if (srcFileInfo->st_size < fileSizeThreshold) { //  Jeżeli rozmiar pliku jest mniejszy niż
wartość progowa
		if (RegularCopy(srcPath, destPath) == -1) { //  Skopiuj używając API linuxa
			syslog(LOG_ERR, "RegularCopy(): Could not copy \"%s\" to \"%s\".", srcPath,
destPath);
			return -1;
		}
	}
	else {  //  jeżeli rozmiar pliku jest większy niż wartość progowa
		if (MmapCopy(srcPath, destPath) == -1) {	//  Skopiuj używając odwzorowania w
pamięci
			syslog(LOG_ERR, "MmapCopy(): Could not copy \"%s\" to \"%s\".", srcPath,
destPath);
			return -1;
		}
	}

	syslog(LOG_INFO, "File \"%s\" has been copied to \"%s\".", srcPath, destPath);
	free(srcFileInfo);
	return 0;
}

/*Funkcja dołącza nazwę pliku filename do ściezki path*/
char *AppendToPath(const char *path, const char *filename) {
	char *newPath = malloc(PATH_MAX * sizeof(char));
	if (sprintf(newPath, "%s/%s", path, filename) < 0) {
		syslog(LOG_ERR, "sprintf(): Could not append \"%s\" to \"%s\".", filename, path);
		return NULL;
	}

	return newPath;
}

/*Funkcja kopiuje wszystkie pliki znajdujące się na liście list do katalogu destDir*/
int CopyAllFilesFromList(List *list, const char *srcDir, const char *destDir) {
	char *fullSrcFilePath = NULL;
	char *fullDestFilePath = NULL;

	Node *current = list->head;
	while (current != NULL) {
		fullSrcFilePath = AppendToPath(srcDir, current->filename);
		fullDestFilePath = AppendToPath(destDir, current->filename);

		if (Copy(fullSrcFilePath, fullDestFilePath) == -1) {
			syslog(LOG_ERR, "Copy(): Could not copy \"%s\" to \"%s\".", fullSrcFilePath,
fullDestFilePath);
			return -1;
		}
```

```
			free(fullSrcFilePath);
			free(fullDestFilePath);
			current = current->next;
		}

		return 0;
}

/*Funkcja usuwa wszystkie pliki znajdujące się na liście list z katalogu path*/
int RemoveAllFilesFromList(List *list, const char *path) {
		char *fullPath = NULL;

		Node *current = list->head;
		while (current != NULL) {
			fullPath = AppendToPath(path, current->filename);
			if (remove(fullPath) == -1) {
				syslog(LOG_ERR, "remove(): \"%s\" (%s)", fullPath, strerror(errno));
				return -1;
			}

			syslog(LOG_INFO, "File \"%s\" has been removed.", fullPath);
			current = current->next;
		}

		free(fullPath);
		return 0;
}

/*Funkcja porównuje czas modyfikacji pliku srcPath oraz destPath*/
int CompareModTime(const char *srcPath, const char *destPath) {
		struct stat *srcFileInfo = NULL;
		struct stat *destFileInfo = NULL;

		srcFileInfo = GetFileInfo(srcPath);
		destFileInfo = GetFileInfo(destPath);

		if (srcFileInfo->st_mtime > destFileInfo->st_mtime) {	//	Jeżeli plik źródłowy był
modyfikowany później niz plik docelowy
			free(srcFileInfo);
			free(destFileInfo);

			return 1;
		}
		else if (srcFileInfo->st_mtime == destFileInfo->st_mtime) {	//	Jeżeli plik źródłowy i
docelowy mają tą samą datę modyfikacji
			free(srcFileInfo);
			free(destFileInfo);

			return 0;
		}
		else {	//	Jeżeli plik źródłowy był modyfikowany wcześniej niz plik docelowy
			free(srcFileInfo);
			free(destFileInfo);

			return -1;
		}
}
```

```
/*Funkcja znajduje plik w liście list o nazwie filename i kopiuje go do destPath jeżeli jego czas
modyfikacji różni się od czasu modyfikacji pliku srcPath*/
int FindAndCopy(List *list, const char *srcPath, const char *destPath, char *filename) {
char *fullSrcFilePath = NULL;
char *fullDestFilePath = NULL;


Node *current = list->head;

while (current != NULL) {
	if (strcmp(current->filename, filename) == 0) {  //  Jeżeli w liście istnieje plik o nazwie
filename
		fullSrcFilePath = AppendToPath(srcPath, filename);
		fullDestFilePath = AppendToPath(destPath, filename);

		if (CompareModTime(fullSrcFilePath, fullDestFilePath) == 1) {	//  Jeżeli plik
źródłowy ma późniejszą date modyfikacji
			if (Copy(fullSrcFilePath, fullDestFilePath) == -1) { //  To skopiuj
				syslog(LOG_ERR,  "Copy():  Could  not  copy  \"%s\"  to  \"%s\".",
fullSrcFilePath, fullDestFilePath);
				return -1;
			}
		}

		RemoveAt(current, list);  //	Usuń z listy

		free(fullSrcFilePath);
		free(fullDestFilePath);
		return 0;
	}
	else {
		current = current->next;
	}
}

return 1;
}

/*Funkcja rekurencyjnie kopiuje katalog path wraz z jego plikami oraz podkatalogami*/
int CopyDirectory(const char *srcPath, const char *destPath) {
	struct dirent *srcFileInfo = NULL;
	DIR *source = NULL;

	struct stat *fileInfo = GetFileInfo(srcPath);

	char *newSrcPath = NULL;
	char *newDestPath = NULL;

	if (mkdir(destPath, fileInfo->st_mode) == -1) {	//	Stwórz nowy katalog
		syslog(LOG_ERR, "mkdir(): \"%s\" (%s)", destPath, strerror(errno));
		return -1;
	}

	syslog(LOG_INFO, "Directory \"%s\" has been created.", destPath);

	source = opendir(srcPath);
	if (!source) {
		syslog(LOG_ERR, "opendir(): \"%s\" (%s)", srcPath, strerror(errno));
		return -1;
	}

	/*Odczytaj zawartość katalogu*/
	while ((srcFileInfo = readdir(source)) != NULL) {
		if (strcmp(srcFileInfo->d_name, ".") == 0 || strcmp(srcFileInfo->d_name, "..") == 0)
{   //  Pomiń katalogi "." i ".."
			continue;
		}
```

```
			newSrcPath = AppendToPath(srcPath, srcFileInfo->d_name);
			newDestPath = AppendToPath(destPath, srcFileInfo->d_name);

			if (srcFileInfo->d_type == DT_REG) { //  Jeżeli jest zwykłym plikiem
				if (Copy(newSrcPath, newDestPath) == -1) {   //  Skopiuj
					syslog(LOG_ERR, "Copy(): Could not copy \"%s\" to \"%s\".", newSrcPath,
newDestPath);
					return -1;
				}
			}
			else if (srcFileInfo->d_type == DT_DIR) {    //  Jeżeli jest katalogiem
				if (CopyDirectory(newSrcPath, newDestPath) == -1) {  // Rekrurencyjnie skopiuj
katalog wraz z jego podkatalogami i plikami
					syslog(LOG_ERR, "CopyDirectory(): Could not copy \"%s\" to \"%s\".",
newSrcPath, newDestPath);
					return -1;
				}

				syslog(LOG_INFO, "Directory \"%s\" has been copied to \"%s\".", newSrcPath,
newDestPath);
			}

			free(newSrcPath);
			free(newDestPath);
		}

		if (closedir(source) == -1) {
			syslog(LOG_ERR, "closedir(): \"%s\" (%s)", srcPath, strerror(errno));
			return -1;
		}

		free(fileInfo);

		return 0;
}

/*Funkcja rekurencyjnie usuwa katalog path wraz z jego plikami oraz podkatalogami*/
int RemoveDirectory(const char *path) {
		DIR *directory = NULL;
		struct dirent *fileInfo = NULL;
		char *newPath = NULL;

		directory = opendir(path);
		if (!directory) {
			syslog(LOG_ERR, "opendir(): \"%s\" (%s))", path, strerror(errno));
			return -1;
		}

		/*Odczytaj zawartość katalogu*/
		while ((fileInfo = readdir(directory)) != NULL) {
			if (strcmp(fileInfo->d_name, ".") == 0 || strcmp(fileInfo->d_name, "..") == 0) { //
Pomiń katalogi "." i ".."
				continue;
			}

			newPath = AppendToPath(path, fileInfo->d_name);
			if (fileInfo->d_type == DT_REG) {    //  Jeżeli jest plikiem
				if (remove(newPath) == -1) {    // Usuń plik
					syslog(LOG_ERR, "remove(): \"%s\" (%s)", newPath, strerror(errno));
					return -1;
				}
			}
			else if (fileInfo->d_type == DT_DIR) {    //  Jeżeli jest katalogiem
				if (RemoveDirectory(newPath) == -1) {   //  Rekurencyjnie usuń podkatalog wraz
z jego podkatalogami
					syslog(LOG_ERR, "RemoveDirectory(): Could not remove %s.", newPath);
					return -1;
				}
			}
		}
```

```
	if (remove(path) == -1) {	//  Usuń katalog
		syslog(LOG_ERR, "remove(): \"%s\" (%s)", path, strerror(errno));
		return -1;
	}

	syslog(LOG_INFO, "Directory \"%s\" has been removed.", path);

	if (closedir(directory) == -1) {
		syslog(LOG_ERR, "closedir(): \"%s\" (%s)", path, strerror(errno));
		return -1;
	}

	free(fileInfo);
	free(newPath);

	return 0;
}

/*Funkcja demonizuje program*/
void Daemonize() {
	int tmp;
	pid_t pid;

	pid = fork(); //	Stwórz nowy proces
	if (pid == -1) {	//  Jeżeli wystąpił błąd
		syslog(LOG_INFO, "fork(): %s", strerror(errno));
		exit(EXIT_FAILURE);
	}
	else if (pid > 0) { //  Jeżeli proces jest rodzicem
		exit(EXIT_SUCCESS); //  Zakończ
	}

	if (setsid() == -1) {	//  Stwórz nową sesję
		syslog(LOG_INFO, "setsid(): %s", strerror(errno));
		exit(EXIT_FAILURE);
	}

	signal(SIGHUP, SIG_IGN);	//  Zignoruj sygnał SIGHUP
	pid = fork();	//  Stwórz nowy proces

	if (pid == -1) {	//  Jeżeli wystąpił błąd
		syslog(LOG_INFO, "fork(): %s", strerror(errno));
		exit(EXIT_FAILURE);
	}
	else if (pid > 0) { //  Jeżeli proces jest rodzicem
		exit(EXIT_SUCCESS); //  Zakończ
	}

	if (chdir("/") == -1) { //  Zmień aktualny katalog na /
		syslog(LOG_INFO, "chdir(): %s", strerror(errno));
		exit(EXIT_FAILURE);
	}

	umask(0);	//  Nadaj maskę do tworzenia plików

	for (int i = 0; i < sysconf(_SC_OPEN_MAX); i++) {	//  Zamknij wszystkie deskryptory
		tmp = close(i);

		if (errno == EBADF) {	// Jeżeli był to zły dekryptor to wyjdź z pętli (zamknięto
wszystkie deskryptory)
			break;
		}

		if (tmp == -1) {
			syslog(LOG_INFO, "%s, %d", strerror(errno), errno);
			exit(EXIT_FAILURE);
		}
	}

	openlog(NULL, LOG_PID, LOG_USER);  //  Otwórz log systemowy
```

```
	if (open("/dev/null", O_RDONLY) == -1) {	//  Otwórz STDIN jako /dev/null
		syslog(LOG_ERR, "STDIN could not be opened properly.");
		exit(EXIT_FAILURE);
	}
	if (open("/dev/null", O_WRONLY) == -1) {	//  Otwórz STDOUT jako /dev/null
		syslog(LOG_ERR, "STDOUT could not be opened properly.");
		exit(EXIT_FAILURE);
	}
	if (open("/dev/null", O_RDWR) == -1) {	//  Otwórz STDERR jako /dev/null
		syslog(LOG_ERR, "STDERR could not be opened properly.");
		exit(EXIT_FAILURE);
	}
}

/*Funkcja synchronizuje katalogi srcPath oraz destPath*/
int SynchronizeDirectories(const char *srcPath, const char *destPath) {
	DIR *source = NULL;
	DIR *destination = NULL;

	struct dirent *srcFileInfo = NULL;
	struct dirent *destFileInfo = NULL;

	List *srcDirFiles = InitList();
	List *destDirFiles = InitList();
	List *srcDirectories = InitList();

	char *newSrcPath = NULL;
	char *newDestPath = NULL;

	Node *nodePtr;

	source = opendir(srcPath);
	if (!source) {
		syslog(LOG_ERR, "opendir(): \"%s\" (%s)", srcPath, strerror(errno));
		return -1;
	}

	destination = opendir(destPath);
	if (!destination) {
		if (errno == ENOENT && recursiveSearch) {	// Jeżeli nie istnieje katalog destPath
i włączona jest rekursywna synchronizacja
			if (CopyDirectory(srcPath, destPath) == -1) {
				syslog(LOG_ERR, "CopyDirectory(): Could not copy \"%s\" to \"%s\".",
srcPath, destPath);
				return -1;
			}

			if (closedir(source) == -1) {
				syslog(LOG_ERR, "closedir(): \"%s\" (%s)", srcPath, strerror(errno));
				return -1;
			}

			DestroyList(srcDirFiles);
			DestroyList(destDirFiles);
			DestroyList(srcDirectories);

			return 0;
		}
		else {
			syslog(LOG_ERR, "opendir(): \"%s\" (%s)", destPath, strerror(errno));
			return -1;
		}
	}
```

```
/*Wczytaj nazwy plików z folderu źródłowego do listy srcDirFiles*/
	while ((srcFileInfo = readdir(source)) != NULL) {
		if (strcmp(srcFileInfo->d_name, ".") == 0 || strcmp(srcFileInfo->d_name, "..") == 0)
{	//	Pomiń katalogi "." i ".."
			continue;
		}

		if (srcFileInfo->d_type == DT_REG) {	//	Jeżeli jest zwykłym plikiem
			Append(srcFileInfo->d_name, srcDirFiles);	//	Dołącz do listy
		}
		else if (srcFileInfo->d_type == DT_DIR && recursiveSearch) {	//	Jeżeli jest
katalogiem i włączona jest opcja rekursywnej synchronizacji
			Append(srcFileInfo->d_name, srcDirectories);	//	Dołącz do listy

			newSrcPath = AppendToPath(srcPath, srcFileInfo->d_name);
			newDestPath = AppendToPath(destPath, srcFileInfo->d_name);
			if (SynchronizeDirectories(newSrcPath, newDestPath) == -1) {	//	Synchronizuj
podkatalogi
				syslog(LOG_ERR, "SynchronizeDirectories(): Could not synchronize \"%s\"
and \"%s\".", newSrcPath, newDestPath);
				return -1;
			}
		}
	}

	/*Wczytaj nazwy plików z folderu docelowego do listy destDirFiles*/
	while ((destFileInfo = readdir(destination)) != NULL) {
		if (destFileInfo->d_type == DT_REG) {	//	Jeżeli jest zwykłym plikiem
			Append(destFileInfo->d_name, destDirFiles);	//	Dołącz do listy
		}
		else if (destFileInfo->d_type == DT_DIR && recursiveSearch) {	// Jeżeli jest
katalogiem
			if (strcmp(destFileInfo->d_name, ".") == 0 || strcmp(destFileInfo->d_name,
"..") == 0) {	//	Pomiń katalogi "." i ".."
				continue;
			}

			if (Contains(srcDirectories, destFileInfo->d_name) == -1) {	//	Jeżeli ten
podkatalog nie znajduje się w katalogu źródłowym
				newDestPath = AppendToPath(destPath, destFileInfo ->d_name);
				if (RemoveDirectory(newDestPath) == -1) {	// To usuń go
					syslog(LOG_ERR, "RemoveDirectory(): Could not remove \"%s\".",
newDestPath);
					return -1;
				}
			}
		}
	}

	/*Kopiuje pliki z katalogu źródłowego do katalogu docelowego jeżeli różni się ich data
modyfikacji*/
	Node *current = srcDirFiles->head;
	int result;
	while (current != NULL) {
		result = FindAndCopy(destDirFiles, srcPath, destPath, current->filename);
		if (result == 0) {	//	Jeżeli został usunięty
			nodePtr = current->next;
			RemoveAt(current, srcDirFiles);	//	Usuń z listy
			current = nodePtr;
		}
		else if (result == -1) {	//	Jeżeli wystąpił błąd
			syslog(LOG_ERR, "FindAndCopy(): Could not copy files from \"%s\" to \"%s\".",
srcPath, destPath);
			return -1;
		}
		else {	//	Jeżeli pliki nie wymagają synchronizacji
			current = current->next;	//	Przejdź do kolejnego pliku
		}
	}
```

```
	/*Skopiuj wszytkie pliki pozostałe w liście srcDirFiles (czyli pliki, których nie ma w katalogu docelowym, ale sa w źródłowym) do katalogu docelowego*/
	if (CopyAllFilesFromList(srcDirFiles, srcPath, destPath) == -1) {
		syslog(LOG_ERR, "CopyAllFilesFromList(): Could not copy files from \"%s\" to \"%s\".", srcPath, destPath);
		return -1;
	}

	/*Usuń wszytkie pliki pozostałe w liście destDirFiles (czyli pliki, których nie ma w katalogu źródłowym, ale są w docelowym) z katalogu docelowego*/
	if (RemoveAllFilesFromList(destDirFiles, destPath) == -1) {
		syslog(LOG_ERR, "RemoveAllFilesFromList(): Could not remove files from \"%s\".", destPath);
		return -1;
	}

	/*Zamknięcie katalogów i zwolnienie pamięci*/
	if (closedir(source) == -1) {
		syslog(LOG_ERR, "closedir(): \"%s\" (%s)", srcPath, strerror(errno));
		return -1;
	}

	if (closedir(destination) == -1) {
		syslog(LOG_ERR, "closedir(): \"%s\" (%s)", destPath, strerror(errno));
		return -1;
	}

	DestroyList(srcDirFiles);
	DestroyList(destDirFiles);
	DestroyList(srcDirectories);
	free(newSrcPath);
	free(newDestPath);

	return 0;
}

/*Funkcja jest handlerem sygnałów*/
void SignalHandler(int signo) {
	switch (signo) {
	case SIGUSR1:
		syslog(LOG_INFO, "Received SIGUSR1. Process awakened by user.");
		break;
	case SIGTERM:
		syslog(LOG_INFO, "Received SIGTERM. Process terminated by user.");
		exit(EXIT_SUCCESS);
		break;
	}
}

int main(int argc, char *const argv[]) {
	const char *appName = strncmp(argv[0], "./", 2) == 0 ? (argv[0] + 2 * sizeof(char)) : argv[0];
	const char *srcPath = argv[1];
	const char *destPath = argv[2];

	struct stat srcDirInfo;
	struct stat destDirInfo;

	if (signal(SIGUSR1, &SignalHandler) == SIG_ERR) {   //  Ustawienie handlera sygnału SIGUSR1
		perror("signal()");
		exit(EXIT_FAILURE);
	}

	if (signal(SIGTERM, &SignalHandler) == SIG_ERR) {   //  Ustawienie handlera sygnału SIGTERM
		perror("signal()");
		exit(EXIT_FAILURE);
	}
```

```
/*pobranie opcjonalnych argumentów*/
	unsigned int argument;
	while ((argument = getopt(argc, argv, "Rs:i:")) != -1) {
		switch (argument) {
		case 's':
			fileSizeThreshold = atoi(optarg);
			break;
		case 'i':
			sleepInterval = atoi(optarg);
			if (sleepInterval > MAX_SLEEP_TIME) {
				sleepInterval = MAX_SLEEP_TIME;
			}

			break;
		case 'R':
			recursiveSearch = true;
			break;
		case '?':
			printf("Wrong Arguments. \n");
			exit(EXIT_FAILURE);
		}
	}

	if (stat(srcPath, &srcDirInfo) == -1) {  //  Pobranie informacji o katalogu srcPath
		printf("\"%s\" does not exist.\n", srcPath);
		exit(EXIT_FAILURE);
	}

	if (stat(destPath, &destDirInfo) == -1) {  //  Pobranie informacji o katalogu destPath
		printf("\"%s\" does not exist.\n", destPath);
		exit(EXIT_FAILURE);
	}

	if (!S_ISDIR(srcDirInfo.st_mode)) {  //  Sprawdzanie czy srcPath jest katalogiem
		printf("\"%s\" is not a directory.\n", srcPath);
		exit(EXIT_FAILURE);
	}

	if (!S_ISDIR(destDirInfo.st_mode)) {  //  Sprawdzanie czy destPath jest katalogiem
		printf("\"%s\" is not a directory.\n", destPath);
		exit(EXIT_FAILURE);
	}

	Daemonize();

	syslog(LOG_INFO,	"%s	started,	RecursiveSearch=%s,	sleepInterval=%ds,
fileSizeThreshold=%dB.",
		appName,
		recursiveSearch ? "true" : "false",
		sleepInterval,
		fileSizeThreshold
	);

	while (1) {
		syslog(LOG_INFO, "Synchronizing directories \"%s\" and \"%s\".", srcPath, destPath);

		if (SynchronizeDirectories(srcPath, destPath) == -1) {
			syslog(LOG_ERR,  "SynchronizeDirectories():  An  error  has  occured.  Process
terminated.");
			exit(EXIT_FAILURE);
		}

		syslog(LOG_INFO, "%s went to sleep for %d seconds.", appName, sleepInterval);
		sleep(sleepInterval);
	}

	exit(EXIT_SUCCESS);
}
```

Opis algorytmów i funkcji

Algorytm synchronizacji katalogów zawiera się w funkcji

```
int SynchronizeDirectories(const char *srcPath, const char *destPath);
```

Opis zostanie podzielony na dwie części, zwykłą wersję algorytmu oraz rekurencyjną.

Algorytm rozpoczyna się od otwarcia katalogów funkcją *opendir()*, następnie do list jednokierunkowych wczytywane są nazwy plików znajdujących się w katalogach. Następnie w pętli *while* za pomocą funkcji *FindAndCopy()* porównywane są nazwy plików z obu list, jeżeli jeden plik znajduje się na obu listach, porównuje się jego daty modyfikacji. Jeżeli data modyfikacji pliku w katalogu źródłowym jest późniejsza niż pliku w katalogu docelowym, plik jest kopiowany. Następnie za pomocą funkcji *CopyAllFilesFromList()* kopiowane są wszystkie pliki pozostałe na liście *srcDirFiles* (czyli pliki, które znajdują się w katalogu źródłowym, ale brak ich w katalogu docelowym) oraz za pomocą funkcji *RemoveAllFilesFromList()* usuwane z katalogu docelowego są wszystkie pliki pozostałe na liście *destDirFiles* (czyli pliki, które znajdują się w katalogu docelowym, ale brak ich w katalogu źródłowym). Na koniec zamykane są wszystkie deskryptory funkcją *closedir()* oraz dealokowana jest pamięć funkcjami *DestroyList()* oraz *free().*

W przypadku rekurencyjnym algorytm nie wiele się różni, gdy nie istnieje katalog docelowy, wywoływana jest funkcja CopyDirectory(), która kopiuje ten katalog wraz zawartością z folderu źródłowego. W przypadku rekurencyjnego algorytmu tworzona jest również trzecia lista, do której wczytywane są nazwy katalogów z katalogu źródłowego, po wczytaniu nazwy katalogu, następuje rekurencyjne wywołanie funkcji SynchronizeDirectories() dla wczytanego katalogu.

Funkcja zwraca 0 w przypadku sukcesu oraz -1 w przypadku błędu.

```
void Daemonize();
```

Demonizacja programu rozpoczyna się od wywołania funkcji *fork(),* która tworzy nowy proces, następnie następuje zakończenie procesu rodzica. Proces dziecko wywołuje funkcję *setsid(),* która tworzy nową sesję. Następnie użyta jest funkcja *signal()* w celu zignorowanie sygnału *SIGHUP* oraz następuje kolejne wywołanie *fork()* i zakończenie procesu rodzica. Następnie funkcja *chdir()* ustawia katalog bieżący na *„/”*, a funkcja *umask()* ustawia maskę tworzenia plików dla procesu. Następnie w pętli for zamykane są wszystkie otwarte deskryptory. Ostatnim krokiem jest otwarcie logu systemowego oraz otworzenie *STDIN*, *STDOUT* oraz *STDERR* jako */dev/null*.

```
int RemoveDirectory(const char *path);
```

Pierwszym krokiem jest otworzenie katalogu funkcją *opendir()*. Nastepnie w pętli *while* jeżeli funkcja *readdir()* odczytała plik, jest on usuwany, jeżeli odczytany został katalog następuje wywołanie funkcji *RemoveDirectory()* dla tego katalogu. Po zakończeniu pętli usuwany jest katalog, zamykany jest deskryptor oraz dealokowana jest pamięć.

Funkcja zwraca 0 w przypadku sukcesu oraz -1 w przypadku błędu.

```
int CopyDirectory(const char *srcPath, const char *destPath);
```

Funkcja *mkdir()* tworzy nowy katalog z uprawnieniami katalogu źródłowego, następnie funkcja *opendir()* otwiera katalog źródłowy. W pętli *while* odczytywana jest zawartość katalogu źródłowego, jeżeli odczytany jest plik, jest on kopiowany, jeżeli doczytany jest katalog następuje wywołanie funkcji *CopyDirectory()* dla tego katalogu. Po zakończeniu pętli zamykany jest deskryptor i dealokowana jest pamięć.

Funkcja zwraca 0 w przypadku sukcesu oraz -1 w przypadku błędu.

`int FindAndCopy(List *list, const char *srcPath, const char *destPath, char *filename);`

Funkcja w pętli *while* szuka pliku *filename* w liście list, jeżeli go znajdzie porównywany jest czas modyfikacji funkcją *CompareModTime(),* jeżeli czas modyfikacji pliku z katalogu źródłowego jest późniejszy niż pliku z katalogu docelowego, jest on kopiowany funkcją *Copy().*

Funkcja zwraca 0 w przypadku sukcesu, 1 w przypadku, gdy pliku nie znaleziono w liście oraz -1 w przypadku błędu.

`int Copy(const char *srcPath, const char *destPath);`

Pobierany są informacje o pliku z katalogu źródłowego za pomocą funkcji *GetFileInfo(),* następnie jeżeli rozmiar pliku jest mniejszy niż *fileSizeThreshold* plik jest kopiowany funkcją *RegularCopy(),* w przeciwnym wypadku plik jest kopiowany za pomocą *MmapCopy().* Na koniec dealokowana jest pamięć.

Funkcja zwraca 0 w przypadku sukcesu oraz -1 w przypadku błędu.

`int RegularCopy(const char *srcPath, const char *destPath);`

Pierwszym krokiem jest otworzenie deskryptorów plików funkcją *open()* oraz pobranie informacji o pliku źródłowym funkcją *GetFileInfo(),* jeżeli plik w katalogu docelowym nie istnieje to jest on tworzony. Następnie w pętli *while* za pomocą funkcji *read()* odczytywane jest *BUFFER_SIZE* Bajtów z pliku źródłowego, po czym są one zapisywane do pliku docelowego funkcją *write().* Na koniec funkcja *SyncModTime()* ustawia czas modyfikacji pliku docelowego na czas modyfikacji pliku źródłowego oraz zamykane są deskryptory.

Funkcja zwraca 0 w przypadku sukcesu oraz -1 w przypadku błędu.

`int MmapCopy(const char *srcPath, const char *destPath);`

Pierwszym krokiem jest otworzenie deskryptorów plików funkcją *open()* oraz pobranie informacji o pliku źródłowym funkcją *GetFileInfo(),* jeżeli plik w katalogu docelowym nie istnieje to jest on tworzony. Następnie za pomocą funkcji *mmap()* tworzone jest odwzorowanie w pamięci pliku źródłowego, następnie funkcja *ftruncate()* ustawia rozmiar pliku docelowego na rozmiar pliku źródłowego po czym również jest on odwzorowywany w pamięci. Następnie funkcja *memcpy()* kopiuje odwzorowanie pliku źródłowego do miejsca odwzorowania pliku docelowego po czym funkcja *munmap()* usuwa odwzorowania. Na koniec funkcja *SyncModTime()* ustawia czas modyfikacji pliku docelowego na czas modyfikacji pliku źródłowego, zamykane są deskryptory oraz dealokowana jest pamięć.

Stałe

`BUFFER_SIZE` – Rozmiar bufora używanego przy kopiowaniu plików używając API linuxa, ustawiony jest na 131072B, czyli 128kB.

`MAX_SLEEP_TIME` – Maksymalny czas uśpienia programu, ustawiony jest na 86400s, czyli 24h.

Zmienne Globalne

`size_t fileSizeThreshold` - Próg rozmiaru pliku wyrażany w bajtach, pliki, których rozmiar jest większy od tej wartości są kopiowane przy użyciu odwzorowania w pamięci. Domyślnie wartość ta wynosi 1048576B, czyli 1MB.

`int sleepInterval` – Czas spania procesu wyrażany w sekundach pomiędzy operacjami synchronizacji. Domyślnie wartość ta wynosie 300s, czyli 5m.

`bool recursiveSearch` – Zmienna ta przyjmuje wartość *true*, jeżeli program został uruchomiony z opcją rekursywnej synchronizacji katalogów. Domyślnie jest ustawiona na *false*.