Betriebssysteme
Threading
Praktikum 6

Fachhochschule Bielefeld
Campus Minden
Studiengang Informatik

Beteiligte Personen:

Name	Matrikelnummer
Peter Dick	1050185

---

25. Mai 2016

Inhaltsverzeichnis

Aufgabe 6 - Multi-Threading und Synchronisation

1 Aufgabenstellung

1. Dem Programm wird per Kommandozeile ein Ordnerpfad übergeben.

2. Arbeiten Sie mit mehreren Threads. Ein Thread (Leser-Thread) liest die Dateien in dem übergebenen Ordner ein und hängt den Inhalt sowie den Dateipfad an eine Queue an. Nutzen Sie dafür eine struct Job.

3. Eine zur Compileteit konfigurierbare Anzahl von Threads (Komprimierungs-Thread) liest jeweils einen Job aus der Queue, komprimiert seinen Inhalt und speichert diesen im Format <alter Dateiname>.compr. Dies soll solange wiederholt werden, bis die Queue leer und der Leser-Thread beendet ist.

4. Der Leser-Thread soll Dateien, die mit .compr enden, ignorieren.

5. Ein Komprimierungs-Thread bekommt bei seiner Erstellung als Parameter eine Instanznummer zugeordnet, die ihn z.B. bei Debug-Ausgaben eindeutig identifiziert.

6. Die Zugriffe auf die Queue müssen synchronisiert, d.h. gegeneinander geschützt sein.

7. Fügen Sie im Leser-Thread nach dem Einlesen einer Datei ein sleep(1) und in den Kompressions-Threads nach den Komprimieren ein sleep(3) ein, um eine langsame Festplatte und einen komplexen Kompressionsalgorithmus zu simulieren. Beobachten Sie, wie ihr Programm mit und ohne die sleep-Anweisungen arbeitet.

8. Bestimmen Sie näherungsweise die Laufzeit bei unterschiedlicher Anzahl von Kompressions-Threads, z. B. mit der Funktion difftime() in main()

2 Aufgabe 6.1

2.1 Vorbereitung

C-Projekt anlegen.
Makefile schreiben.
Include-Dateien includieren.

2.2 Durchführung

Code schreiben und dann testen bzw debuggen.

2.3 Fazit

```
int main(int argc, char *argv[]) {
        int theads, cauntthreads = 0, statuskt[cauntthreads];
        if(argc < 2) {
                fprintf(stderr, "Zu wenig Komandozeilenargumente.");
                return EXIT_FAILURE;
        } else if (argc == 2) {
                theads = 3;
        }
        theads = atoi(argv[2]);
        pthread_t lthread, kthread[theads];
        joblist.jobs = queue_create();
        char *dirname = argv[1];
```

Der Ordnerpfad ist das zweite argument das der Main-Methode übergeben wird. Das Programm wird mit dem Befehl ./main <Verzeichnis> gestartet

3 Aufgabe 6.2

3.1 Vorbereitung

```
typedef struct job {
        char * inhalt;
        char * dateipfad;
} Job;

typedef struct joblist {
        pthread_mutex_t *mutex;
        Queue jobs;
} Joblist;
```

struct Job und Joblist schreiben.

3.2 Durchführung

Code schreiben und dann testen bzw debuggen.

3.3 Fazit

```
void* readThread(void *arg) {
        FILE *file;
        char* dirname = (char*)arg;
        DIR *dir = opendir(dirname);
        struct dirent *dirptr;
        Job *job  = (Job*) malloc(sizeof(Job));
        if(!job) {
                fprintf(stderr,"Fehler beim Speicher reseviren.");
                return (void*) EXIT_FAILURE;
        }
        if(dir == NULL) {
                fprintf(stderr,"Fehler beim oeffnen der Datei.");
                return (void*) EXIT_FAILURE;
        }
        dirptr = readdir(dir);
        while(dirptr != NULL) {
                char *filename = (*dirptr).d_name;
                char *dateiendung = strrchr(filename, '.');
                if(strcmp(filename, ".") && strcmp(filename, "..") && strcmp(dateiendung, ".compr")) {
                        file = fopen(filename, "r");
                        if(file == NULL) {
                                fprintf(stderr, "Datei konnte nicht geoeffnet werden\n.");
                        } else {
                                job->dateipfad = filename;
                                //int i = fscanf(file,"%s", job->inhalt);
                                //int *temp = "";
                                char *line = NULL;
                                size_t *buffersize = 0;
                                int i = getline(&line,buffersize,file);
                                while(i != -1) {
                                        getline(&line,buffersize,file);
                                        strcat(job->inhalt, line);
                                }
                                int status=pthread_mutex_lock(joblist.mutex);
                                queue_insert(joblist.jobs, job);
                                status=pthread_mutex_unlock(joblist.mutex);
                                printf("%s\n" , filename);
                                fclose(file);
                        }
                }
                dirptr = readdir(dir);
                job->dateipfad = "";
                job->inhalt = "";
        }
        free(job);
        if (closedir(dir) == -1) {
                fprintf(stderr,"Fehler beim Schliessen von %s\n", dirname);
        }
        return (void*) EXIT_SUCCESS;
}
```

Zuerst wird die Thread-Methode für den Leser erstellt. Mit opendir wird das übergebene Verzeichnis geöffnet. Mit readdir wird das Verzeichnis gelesen. Mit "char *filename = (*dirptr).d_name;" wird der Name der Datei ermittelt. Die wird dann mit fopen geöffnet und danach mit getline Zeilenweise eingelesen. Die ermittelten Daten werden in der struct Job gespeichert und dann mit "queue_insert(joblist.jobs, job);" der Queue hizugefügt. Dann werden mit pthread_create in der main Methode die Threads erstellt und mit pthread_join auf sie gewartet.

4 Aufgabe 6.3

4.1 Vorbereitung

keine

4.2 Durchführung

Code schreiben und dann testen bzw debuggen.

4.3 Fazit

```
int main(int argc, char *argv[]) {
        int theads, cauntthreads = 0, statuskt[cauntthreads];
        if(argc < 2) {
                fprintf(stderr, "Zu wenig Komandozeilenargumente.");
                return EXIT_FAILURE;
        } else if (argc == 2) {
                theads = 3;
        }
        theads = atoi(argv[2]);
```

Mit hilfe eines dritten Kommandozeilenparameters kann die Anzahl der Kommrimier-Threads zur Compilezeit festgelegt werden. Wenn der dritte Kommandozeilenparameter fehlt wird ein Standardwert festgelegt.

```
void* kompThread(void *arg) {
        char* dirname = (char*)arg;
        DIR *dir = opendir(dirname);
        Result *result;
        if(dir == NULL) {
                fprintf(stderr,"Fehler beim oeffnen der Datei.");
                return (void*) EXIT_FAILURE;
        }
        Job *job  = (Job*) malloc(sizeof(Job));
                if(!job) {
                        fprintf(stderr,"Fehler beim Speicher reseviren.");
                        return (void*) EXIT_FAILURE;
                }
        while(joblist.jobs->head != 0 || statuslt != 0) {
                int status=pthread_mutex_lock(joblist.mutex);
                job = queue_head(joblist.jobs);
                status=pthread_mutex_unlock(joblist.mutex);
                queue_delete(joblist.jobs);
                result = compress_string(job->inhalt);
                char *newName = "";
                strcpy(newName, job->dateipfad);
                strcat(newName, ".compr");
                if( (rename(job->dateipfad,newName)) < 0) {
                        fprintf(stderr, "Fehler beim Umbenennen von %s", job->dateipfad);
                        return (void*)EXIT_FAILURE;
                }
                FILE* fp;
                fp = fopen(job->dateipfad,"w");
                fprintf(fp, result->data);
                fclose(fp);
        }
        free(job);
        free(result->data);
        free(result);
        if (closedir(dir) == -1) {
                fprintf(stderr,"Fehler beim Schliessen von %s\n", dirname);
        }
        return (void*) EXIT_SUCCESS;
}
```

Zuerst wird eine Thread-Methode fürs Komprimieren erstellt. Mit opendir wird das übergebene Verzeichnis geöffnet. Dann wird mit "job = queue_head(joblist.jobs);" die erste Datei aus der Queue gelolt und danach aus der Queue gelöscht. Dann wird der Inhalt vom job mit compress_string Komprimiert. Dann wird die Datei dessen Inhalt grade Bearbeitet wurde mit rename in <alter Name>.compr umbenannt. Danach wird der Kommprimierte String mit fprintf in die Umbenannte Datei geschrieben. Mit "while(joblist.jobs->head != 0 || statuslt != 0)" (statuslt ist der Status des Lese-Threads) wird sicher gestllt dass der Thread sollange arbeitet bis die Queue leer ist und der Lese-Thread beendet ist.

5 Aufgabe 6.4

5.1 Vorbereitung

keine

5.2 Durchführung

Code schreiben und dann testen bzw debuggen.

5.3 Fazit

```
char *filename = (*dirptr).d_name;
char *dateiendung = strrchr(filename, '.');
if(strcmp(filename, ".") && strcmp(filename, "..") && strcmp(dateiendung, ".compr")) {
```

Mit der Methode strrchr wird die Dateiendung ermittelt. Die wird dann mit strcmp auf die endung ".compr" verglichen wenn der Vergleich true ergibt dann wird die Datei übersprungen.

6 Aufgabe 6.5

6.1 Vorbereitung

keine

6.2 Durchführung

Code schreiben und dann testen bzw debuggen.

6.3 Fazit

Die Komprimierungs-Threads sind in einen Array gespeichert und haben einen eindeutigen Index.

7 Aufgabe 6.6

7.1 Vorbereitung

keine

7.2 Durchführung

Code schreiben und dann testen bzw debuggen.

7.3 Fazit

Die Queue ist mit einen Mutex geschützt der mit "pthread_mutex_init(joblist.mutex, NULL);" initialisiert wird. Mit "pthread_mutex_lock(joblist.mutex);" in den Thread-Methoden die Queue gesperrt wird und mit pthread_mutex_unlock wider entsperrt wird. Und schliesslich mit "pthread_mutex_destroy(joblist.mutex);" zerstört wird.

8 Aufgabe 6.7

8.1 Vorbereitung

keine

8.2 Durchführung

Code schreiben und dann testen bzw debuggen.

8.3 Fazit

9 Aufgabe 6.8

9.1 Vorbereitung

keine

9.2 Durchführung

Code schreiben und dann testen bzw debuggen.

9.3 Fazit