



Programmieraufgabe

Kurs: Betriebssysteme
Dozent: Prof. Dr. Arthur Zimmermann

Abgabe:

Die Textdateien mit C-Quellcode und die Testdateien in eine ZIP-Datei verpacken und bis zum festgelegten Datum im Moodle hochladen. Diese ZIP-Datei muss nach dem folgenden Muster (ohne Leerzeichen) benannt werden:

IhrNachname.zip

Email: *arthur.zimmermann@hwr-berlin.de*

Viel Erfolg !

Aufgabe – Ein vorgegebenes Muster in mehreren Textdateien parallel suchen (multithreaded search)

Erhöhung der Leistung eines Betriebssystems mit konventionellen Mitteln ist weiter so gut wie unmöglich: die Übertragungsgeschwindigkeit nähert sich der vom Licht, und die Abstände zwischen den elektronischen Komponenten sind schon im Nano-Bereich.

Ein der Lösungswege, der eine Leistungserhöhung verspricht, ist die parallele Verarbeitung. Man muss dabei die Denkweise bei dem Entwurf der Algorithmen ändern. Obwohl mehrere technische Engpässe existieren, hier geht es nur um Algorithmus und seine Implementierung.

Man bereitet etwa 20 Textdateien für Test in einem Verzeichnis (z.B. – [./ToSearch](#)) vor. Die Dateien müssen nicht unbedingt ganz groß sein. In diesen Dateien wird es nach einem vorgegebenen Muster mit Ihrem C-Programm gesucht. Die Komponenten dieses C-Programms tun die Suche gleichzeitig (parallel). Die Anzahl und die Namen der Dateien dürfen nirgendwo im Programm festgelegt werden. [2]

Das Programm baut eine einfach verkettete Liste mit den Dateinamen auf (als globale Variable). Das Programm startet mehrere Threads, die diese Liste abarbeiten und das Muster in den Dateien suchen. Anzahl der Threads (4 bis 8), wie auch Suchmuster, sollte im Programm festgelegt werden. Alle Threads führen eine und die selbe Funktion aus.

Folgende Funktionen/Anforderungen werden bewertet:

Die Anwendung sollte nach dem Schema aus Übungen aufgebaut werden: eine *makefile*-Datei, eine Header-Datei, eine Datei mit *main()*-Funktion, eine Datei mit Thread-Funktion und eine Datei mit anderen Funktionen. [5]

Die *makefile*-Datei muss Ihre eigene sein (nicht die allgemeine aus unserem Unterricht), und die Befehle *make*, *make run* und *make clean* müssen funktionieren. [3]

In der Header-Datei müssen alle notwendigen *include* stehen, Definitionen von Typen, globalen Konstanten und/oder globalen Variablen (wie z.B. Anfangsadresse der Liste), Präprozessorvariablen (wie z.B. das zu suchende Muster), sowie die Prototypen. [5]

Die einfach verkettete Liste basiert auf einem Datentyp (Knoten), der folgende Komponenten enthält:

- * den Name der Datei;
- * das Mutex für diese Datei;
- * die Nummer des Threads;
- * die ganzzahlige Variable (=1 wenn Suchmuster gefunden wurde, =0 sonst, als Defaultwert);
- * den Verweis (Adresse) auf den nächsten Knoten.

Die Länge des Namens der Datei kann, einfacheitshalber, beschränkt werden, z.B. auf 8 Symbole (ohne Leerzeichen, ohne spezielle Symbole). Als Nummer des Threads wird im Laufe des Algorithmus die Nummer von dem Thread eingetragen, der diese Datei für sich reserviert hat, momentan durchsucht oder schon durchsucht hat. Der Anfangswert (Initialisierungswert) dafür ist eine 0. [4]

init() Diese Funktion initialisiert die globalen Variablen (z.B. die Anfangsadresse der Liste) und ruft die Funktion *GenList()* auf. Die Funktion *init()* hat keine Parameter und keine Rückgabewerte. [3]

finish() Diese Funktion gibt den vorher reservierten Speicherplatz der Knoten der einfach verketteten Liste frei. Zuvor wird hier aber die Funktion *ShowList()* aufgerufen. Die Funktion *finish()* hat keine Parameter und keine Rückgabewerte. [5]

GenList() Diese Funktion öffnet das Verzeichnis mit vorbereiteten Textdateien (z.B. – [./ToSearch](#)), liest alle Dateinamen aus und schließt Verzeichnis zu. Für jede Datei (selbstverständlich außer `.` und `..`) ruft sie die Funktion *Add2List()* auf und übergibt ihr einen (nur einen!) Dateinamen. Diese Funktion hat keine Parameter und keine Rückgabewerte. [8]

Add2List() Diese Funktion bekommt als Parameter den Namen einer Datei, reserviert den Speicherplatz für einen Knoten, deren Datentyp oben beschrieben ist, initialisiert diesen einen Knoten und fügt ihn in die Liste hinzu. Also, diese Funktion arbeitet mit einem (neuen) Knoten und muss nicht die ganze Liste auf einmal aufbauen. Sie baut die Liste auf, indem sie mehrmals aus *GenList()* aufgerufen wird. Diese Funktion hat keine Rückgabewerte. [9]

Search() Diese Funktion bekommt als Parameter den Namen einer Datei, öffnet diese Datei zum Lesen im Verzeichnis (z.B. – [./ToSearch](#)), liest den Inhalt der Datei Zeile für Zeile aus und sucht in jeder Zeile nach dem vorgegebenen Muster. Danach wird Datei geschlossen. Diese Funktion gibt den ganzzahligen Wert 1 zurück, wenn Muster gefunden wurde, und den Wert 0 anderenfalls. [8]

ShowList() Diese Funktion zeigt alle Knoten der Liste auf dem Bildschirm an. Sie hat keine Parameter und keine Rückgabewerte. Es lohnt sich z.B. alle Komponenten jedes Knotens anzuzeigen. Diese Funktion hat mit der Freigabe des Speicherplatzes nichts zu tun. [4]

ThrdFunc() Diese Funktion wird in jedem Thread verwendet. Sie muss die standardmäßigen Parameter und den standardmäßigen Rückgabewert haben, sonst funktioniert sie nicht. Sie geht nur einmal die ganze verkettete Liste der Dateien durch. Für jeden Knoten macht sie folgendes:

Sie überprüft, ob die Threadnummer in dem Knoten 0 oder nicht 0 ist. Die 0 bedeutet, dass kein Thread diesen Knoten für Bearbeitung (für Suchen) reserviert hat. Keine 0 (d.h. eine positive Zahl) bedeutet, dass der Thread mit dieser Nummer den Knoten für sich reserviert hat und jetzt grade die entsprechende Datei durchsucht, oder der Suchvorgang schon zu Ende ist.

In dem Fall, wenn die Threadnummer in einem Knoten die 0 ist, versucht die Funktion den Knoten mit Mutex zu blockieren – wenn es klappt, dann schreibt sie eigene Threadnummer in die Komponente der Liste, macht den Mutex frei, ruft die Funktion *Search()* auf und übergibt ihr den Namen der Datei aus dem Knoten. Wenn die Funktion *Search()* abgearbeitet hat, dann wird deren Rückgabewert in den entsprechenden Knoten der Liste geschrieben, was eigentlich bedeutet, dass das Suchmuster gefunden wurde.

Klappt das Blockieren mit Mutex nicht, darf sie nicht auf die Freigabe des Mutex warten – sie meldet in der Protokolldatei, dass entsprechender Knoten (Dateiname) schon blockiert ist, und übergeht einfach zum nächsten Knoten.

In dem Fall, wenn die Threadnummer in dem Knoten nicht 0 ist, meldet Thread, dass die entsprechende Datei in Bearbeitung durch entsprechenden Thread ist, und übergeht ebenfalls zum nächsten Knoten.

In allen drei Fällen muss diese Funktion in der Protokolldatei eine Meldung schreiben. Jeder Thread hat eine eigene Protokolldatei. [15]

main() Diese Funktion organisiert ein Array von fester Größe, der als Komponenten die Thread-Informationen (*pthread_t*) beinhaltet. Die Länge des Arrays wird in der Header-Datei als Preprocessorvariable festgelegt (wie gesagt – 4 bis 8, eigentlich, das ist die Anzahl von Threads). Die Hilfsarrays können hier auch organisiert werden. Diese Funktion ruft am Anfang die Funktion *init()* und am Ende die Funktion *finish()* auf. Dazwischen werden zwei Schleifen programmiert: in der ersten werden alle Threads gestartet, in der zweiten wird es auf das Ende von Threads gewartet. Beim Starten führt jeder Thread die Funktion *ThrdFunc()* aus, wobei die fortlaufende Nummer des Threads als Parameter an *ThrdFunc()* übergeben wird (1, 2, 3, ...). [9]

Bewertung von diesen Funktionen/Prozeduren: insgesamt 80 Punkte (40% der Gesamtnote).