Inhaltsverzeichnis

[1 Aufgabenstellung 3](#__RefHeading___Toc525_1561136669)

[2 Realisierung 3](#__RefHeading___Toc527_1561136669)

[2.1 Allgemein 3](#__RefHeading___Toc529_1561136669)

[3 C-Programm 3](#__RefHeading___Toc533_1561136669)

[3.1 Compilieren 3](#__RefHeading___Toc535_1561136669)

[3.2 Ausführen 3](#__RefHeading___Toc537_1561136669)

[4 Ergebnis 4](#__RefHeading___Toc539_1561136669)

[5 Code 4](#__RefHeading___Toc1826_1561136669)

[5.1 Beschreibung 4](#__RefHeading___Toc1828_1561136669)

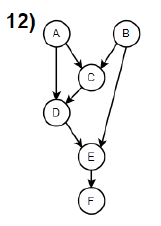
[5.2 Code 4](#__RefHeading___Toc1830_1561136669)

# Aufgabenstellung

Ein Graph soll mit der entsprechenden Katalognummer mithilfe von parallelen Threads und geeigneten Synchronisationsmitteln realisiert werden.

* Die einzelnen Buchstaben repräsentieren die einzelnen Prozesse.
* Die Pfeile repräsentieren die Abfolge und Abhängigkeiten.
* Realisieren sie jeden Prozess durch eine zufällige Zeitspanne mit geeigneten
* Ausgaben.
* Die möglichen Abfolgen der einzelnen Prozesse sind zu gewährleisten

Da meine Katalognummer 12 ist, ist folgender Graph zu realisieren:



# Realisierung

## Allgemein

Die Entscheidung für das Synchronisationsmittel fiel auf Mutex, da damit jeder einzelne Buchstabe/Prozess abgebildet werden kann (entweder noch nicht fertig = locked oder fertig = unlocked). Der Code wurde auf Manjaro mit KDE Plasma getestet.

# C-Programm

## Compilieren

Der Code kann mit dem gcc-Compiler compiliert werden. Dazu gibt man folgenden Befehl ein,

wenn das .c-File mutex.c heißt:

$ gcc -pthread -o mutex mutex.c

jedoch muss je nach Linux-Art noch gcc nachinstalliert werden:

$ sudo apt install build-essential

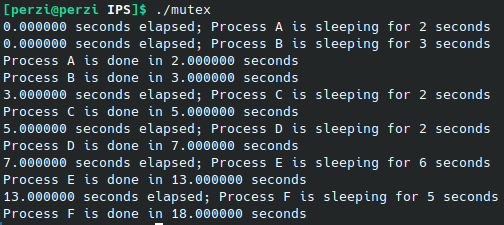
## Ausführen

Anschließend kann man dieses mit

$ ./mutex

ausführen.

# Ergebnis



x.xxxxx seconds elapsed = … done in x.xxxxx seconds = Zeit die seit dem Ausführen des Codes vergangen ist

...sleeping for x seconds = Zeit die der jeweilige Prozess schläft

Die Threads starten alle (ziemlich) gleichzeitig. Jedoch warten verschiedene Threads mittels pthread\_mutex\_lock() auf andere Threads, die noch nicht fertig sind. Sobald ein Thread fertig ist, wird mittels pthread\_mutex\_unlock() der entsprechende Mutex freigegeben. Die entsprechenden folgenden Threads können dann beginnen/weitermachen.

Wie auch bei der Aufgabenstellung zu sehen: Prozess A und B starten gleichzeitig. Prozess C startet erst, wenn Prozess A und B fertig sind. Prozess D startet erst, wenn Prozess A und C fertig sind. Prozess E startet erst, wenn Prozess B und D fertig sind. Prozess F startet erst, wenn Prozess E fertig ist. Die Abhängigkeit wird hier auch mittels der vergangenen Zeit angezeigt. So startet Prozess C erst nachdem Prozess A und B fertig sind und nicht nachdem nur Prozess A fertig ist. Der aktuelle Prozess ist (ungefähr) immer nach der Fertigstellungszeit des vorherigen Prozesses + der sleep-Zeit des aktuellen Prozesses fertig.

# Code

## Beschreibung

Für jeden Prozess (A-F) gibt es einen Mutex. Diese werden alle am Anfang auf „locked“ gesetzt.

Anschließend werden die Prozesse A-F gestartet. Da A und B wie im Diagramm beschrieben sofort starten sollen, warten diese 2 Prozesse nicht auf einen anderen Mutex/Prozess.

Mit der sleep()-Funktion wird der „eigentliche Prozess“ simuliert. Da diese unterschiedlich lange dauern können, ist dieser immer unterschiedlich/zufällig lange.

Die anderen Prozesse warten auf einen Mutex/Prozess, bevor sie starten. Sobald ein Prozess fertig ist, wird der eigene Mutex freigegeben es kann (wenn so vorgesehen) ein anderer Prozess starten.

Mit start, end, end2 und den time()-Aufrufen wird die Zeit gemessen, die für die einzelnen Prozesse seit dem beginn aller Prozesse benötigt haben.

## Code

#include <pthread.h>

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <time.h>

// mutexes A-F as PTHREAD\_MUTEX\_INITIALIZER (do not need to be initialized)

pthread\_mutex\_t mutexA = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_mutex\_t mutexB = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_mutex\_t mutexC = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_mutex\_t mutexD = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_mutex\_t mutexE = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_mutex\_t mutexF = PTHREAD\_MUTEX\_INITIALIZER;

// time variables for time elapsed

time\_t start, end, end2;

// function to delay and unlock a mutex

void delay\_and\_unlock\_mutex(pthread\_mutex\_t \*mutex, char \*process\_name)

{

// delay for a random amount of time between 2 and 6 seconds

int delay = rand() % 5 + 2;

// get the current time

time(&end);

printf("%f seconds elapsed; %s is sleeping for %d seconds \n", difftime(end, start), process\_name, delay);

// sleep for the random amount of time

sleep(delay);

// unlock the current mutex

pthread\_mutex\_unlock(mutex);

// get the current time

time(&end2);

printf("%s is done in %f seconds \n", process\_name, difftime(end2, start));

}

// process A

void \*processA(void \*arg)

{

delay\_and\_unlock\_mutex(&mutexA, "Process A");

**return** NULL;

}

// process B

void \*processB(void \*arg)

{

delay\_and\_unlock\_mutex(&mutexB, "Process B");

**return** NULL;

}

// process C

void \*processC(void \*arg)

{

// wait for mutex A and B to be unlocked

pthread\_mutex\_lock(&mutexA);

pthread\_mutex\_unlock(&mutexA);

pthread\_mutex\_lock(&mutexB);

pthread\_mutex\_unlock(&mutexB);

delay\_and\_unlock\_mutex(&mutexC, "Process C");

**return** NULL;

}

// process D

void \*processD(void \*arg)

{

// wait for mutex A and C to be unlocked

pthread\_mutex\_lock(&mutexA);

pthread\_mutex\_unlock(&mutexA);

pthread\_mutex\_lock(&mutexC);

pthread\_mutex\_unlock(&mutexC);

delay\_and\_unlock\_mutex(&mutexD, "Process D");

**return** NULL;

}

// process E

void \*processE(void \*arg)

{

// wait for mutex B and D to be unlocked

pthread\_mutex\_lock(&mutexB);

pthread\_mutex\_unlock(&mutexB);

pthread\_mutex\_lock(&mutexD);

pthread\_mutex\_unlock(&mutexD);

delay\_and\_unlock\_mutex(&mutexE, "Process E");

**return** NULL;

}

// process F

void \*processF(void \*arg)

{

// wait for mutex C and D to be unlocked

pthread\_mutex\_lock(&mutexE);

pthread\_mutex\_unlock(&mutexE);

delay\_and\_unlock\_mutex(&mutexF, "Process F");

**return** NULL;

}

int main()

{

srand(time(0)); // seed the random number generator

// lock all mutexes

pthread\_mutex\_lock(&mutexA);

pthread\_mutex\_lock(&mutexB);

pthread\_mutex\_lock(&mutexC);

pthread\_mutex\_lock(&mutexD);

pthread\_mutex\_lock(&mutexE);

pthread\_mutex\_lock(&mutexF);

time(&start); // get the current time

// create processes

pthread\_t threadA, threadB, threadC, threadD, threadE, threadF;

pthread\_create(&threadA, NULL, processA, NULL);

pthread\_create(&threadB, NULL, processB, NULL);

pthread\_create(&threadC, NULL, processC, NULL);

pthread\_create(&threadD, NULL, processD, NULL);

pthread\_create(&threadE, NULL, processE, NULL);

pthread\_create(&threadF, NULL, processF, NULL);

pthread\_join(threadA, NULL);

pthread\_join(threadB, NULL);

pthread\_join(threadC, NULL);

pthread\_join(threadD, NULL);

pthread\_join(threadE, NULL);

pthread\_join(threadF, NULL);

// destroy mutexes

pthread\_mutex\_destroy(&mutexA);

pthread\_mutex\_destroy(&mutexB);

pthread\_mutex\_destroy(&mutexC);

pthread\_mutex\_destroy(&mutexD);

pthread\_mutex\_destroy(&mutexE);

pthread\_mutex\_destroy(&mutexF);

**return** 0;

}