Fachhochschule Nordwestschweiz
Hochschule für Technik

System-Programmierung (syspr) 04. Juni 2020 thomas.amberg@fhnw.ch

Assessment

Vorname / Name: *(via GitHub)* Punkte: ___ / 80, Note: ___

Klasse: 4ibb2 *Frei lassen für Korrektur.*

Hilfsmittel:

- Aufgaben werden allein zu Hause am Computer gelöst.
- Alle Unterlagen (Slides, Bücher) sind erlaubt, im Sinn von *open book*.
- Plus online Zugriff auf die Linux man pages unter http://man7.org/linux/man-pages

Nicht erlaubt:

- Jegliche Kommunikation mit anderen Personen.

Bewertung:

- Offene Fragen: Bewertet wird Korrektheit, Vollständigkeit und Kürze der Antwort.
- Programme: Bewertet wird die Skizze/Idee und Umsetzung des Programms.

Fragen zur Prüfung:

- Während der Prüfung werden vom Dozent keine Fragen zur Prüfung beantwortet.
- Ist etwas unklar, machen Sie eine Annahme und notieren Sie diese auf der Prüfung.

Bearbeiten der Aufgaben:

- Lesen Sie die Aufgabenstellungen in dieser PDF Datei.
- Bearbeiten Sie die bestehenden *C* und *TXT* Dateien.
- Compilieren Sie die bestehenden *C* Dateien mit

```
$ make FILENAME_OHNE_C
```

z.B. um die Datei hello.c zu compilieren

```
$ make hello
```

Ein Makefile ist bereits vorhanden.

Abgabe via GitHub:

- Committen Sie alle Änderungen an bestehenden *C* und *TXT* Dateien mit

```
$ git commit -m "update" *.c
$ git commit -m "update" *.txt
```

- Übermitteln Sie alle lokalen Commits an GitHub mit

```
$ git push
```

- Es zählt der Stand *auf GitHub* beim letzten Commit vor dem / am Ende der Prüfung.

0: name.txt

Fügen Sie Ihren *Vornamen* und *Namen* in die Datei *name.txt* ein. Verbindlich

1: leet.c

Ergänzen Sie das Programm *leet.c* so, dass es die übergebenen Command Line Argumente zu "Leet Speak" übersetzt, indem es Kleinbuchstaben wie im Beispiel austauscht. Punkte: ___ / 12

```
$ ./leet abcdefghijklmnopqrstuvwxyz
48cd3f9h1jk1mn0pq257uvwxy2
$ ./leet hello hacker
h3110 h4ck32
```

Verwenden Sie dazu die folgenden System Calls:

```
printf(), strlen().
```

Fügen Sie Ihre Idee (kurz) und Ihren Code in die Datei leet.c ein.

2: malloc.c

Ergänzen Sie das Programm *malloc.c* so, dass es so viel Heap Speicher am Stück alloziert, wie möglich, in maximal 1 Sekunde. Geben Sie die allozierte Anzahl Bytes aus. Punkte: ___ / 12

```
$ ./malloc
268435123
```

Verwenden Sie dazu die folgenden System Calls, Fehlerbehandlung nur soweit wie nötig:

```
clock(), free(), malloc(), printf().
```

Fügen Sie Ihre Idee (kurz) und Ihren Code in die Datei malloc.c ein.

3: lastline.c

Ergänzen Sie das Programm *lastline.c* so, dass es die letzte Zeile einer Datei ausgibt. Am Ende einer Zeile steht jeweils ein *\n*, ausser bei der Letzten. Der Dateiname wird als Command Line Argument übergeben. Punkte: ___ / 16

```
$ printf "cash rules\neverything\naround me" > poem.txt
$ ./lastline poem.txt
around me
```

(Fortsetzung auf der nächsten Seite)

Verwenden Sie dazu die folgenden System Calls, Fehlerbehandlung können Sie weglassen:

```
lseek(), open(), read(), write().
```

Fügen Sie Ihre Idee (kurz) und Ihren Code in die Datei lastline.c ein.

4: stack.c

Die Funktion *f()* im Programm *stack.c* ruft sich selbst auf, was zu einem *Stack Overflow* führt. Bei jeder Rekursion wird die Tiefe des Stacks ausgegeben. Ändern Sie das Programm so, dass der Stack Overflow in einem Child Prozess passiert, und die Tiefe des Stacks *i* per Pipe an den Parent gesendet wird. Dieser soll dann nur den letzten Wert von *i* ausgeben. Punkte: ___ / 16

```
$ ./stack
524118
```

Verwenden Sie dazu die folgenden System Calls, Fehlerbehandlung können Sie weglassen:

```
close(), fork(), pipe(), printf(), read(), wait(), write().
```

Fügen Sie Ihre Idee (kurz) und Ihren Code in die Datei stack.c ein.

5: mutex.txt

Wieso kann dieses Programm zu einem *Deadlock* (Stillstand) führen? Punkte: ___ / 8

```
 1: #include <pthread.h>
 2: #include <stdio.h>
 3: #include <unistd.h>
 4:
 5: pthread_mutex_t m0 = PTHREAD_MUTEX_INITIALIZER;
 6: pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER;
 7:
 8: volatile int i;
 9:
10: void *start() {
11:     while (1) {
12:         pthread_mutex_lock(&m1);
13:         pthread_mutex_lock(&m0);
14:         printf("%d\n", i++);
15:         pthread_mutex_unlock(&m0);
16:         pthread_mutex_unlock(&m1);
17:     }
18:     return 0;
19: }                          // Fortsetzung auf der nächsten Seite >>
```

```
20:
21: int main() {
22:     pthread_t t;
23:     pthread_create(&t, NULL, start, NULL);
24:     while (1) {
25:         pthread_mutex_lock(&m0);
26:         pthread_mutex_lock(&m1);
27:         printf("%d\n", i++);
28:         pthread_mutex_unlock(&m1);
29:         pthread_mutex_unlock(&m0);
30:     }
31:     return 0;
32: }
```

Antworten Sie so genau wie möglich in der Datei *mutex.txt*.

6: semrun.c

Ergänzen Sie das Programm *semrun.c* so, dass es max. 3 laufende Instanzen von sich zulässt, mit einer Named Semaphore */semrun*, die beim ersten Aufruf desselben Programms erstellt wird. Die Arbeit des Programms soll durch *sleep(5)* simuliert werden. Punkte: ___ / 16

```
$ ./semrun &
semrun: sleeping for 5s...
$ ./semrun &
semrun: sleeping for 5s...
$ ./semrun &
semrun: sleeping for 5s...
$ ./semrun &
semrun: too many instances
[4] Exit
```

Verwenden Sie dazu die folgenden System Calls, Fehlerbehandlung nur soweit wie nötig:

```
exit(), printf(), sem_close(), sem_init(), sem_open(), sem_post(),
sem_trywait(), sleep().
```

Hinweis 1: sem_open() mit O_EXCL liefert SEM_FAILED, falls Semaphore bereits existiert.
Hinweis 2: sem_trywait() liefert -1, EAGAIN, falls die Semaphore "keinen Platz frei" hat.
Fügen Sie Ihre Idee (kurz) und Ihren Code in die Datei semrun.c ein.