Aufgabe 2: hupsi - Highly Unreliable Parallel Software Igniter (59 Punkte)

Sie dürfen diese Seite zur besseren Übersicht bei der Programmierung heraustrennen!

Schreiben Sie ein POSIX-1.2008 und C11-konformes Programm *hupsi*, das Befehle zeilenweise (ein Befehl pro Zeile, Zeilen mit `'\n'` getrennt) von der Standardeingabe einliest und parallel ausführt. Die maximale Anzahl n an parallel ausgeführten Befehlen soll dabei als Befehlszeilenargument übergeben werden.

Beispielhafter Aufruf von *hupsi* (4 Befehle, davon maximal 2 parallel):

```
chris@legio:~$ ./hupsi 2 < befehle.txt
```

Beispielinhalt befehle.txt:

```
gcc hupsi.c -o hupsi
sleep 5
find / -name bs-klausur.tex
wsort klausurnoten.txt
```

Das Programm soll folgendermaßen strukturiert sein:

- Funktion `main()`: Prüft zunächst die Befehlszeilenargumente und initialisiert ggf. benötigte Datenstrukturen. Nutzen Sie zum Umwandeln der Obergrenze n die Funktion `strtol`, um eine saubere Fehlerprüfung zu ermöglichen.
  Im Anschluss daran werden die auf der Standardeingabe übergebenen Befehle durch Aufruf der nachfolgend beschriebenen `run`-Funktion parallel ausgeführt. Dabei sollen, solange noch nicht ausgeführte Befehle vorhanden sind, stets genau n Befehle parallel laufen. Falls die maximale Zahl an parallel laufenden Prozessen erreicht ist, soll mittels der unten beschriebenen Funktion `waitProcess()` auf das Terminieren eines zuvor gestarteten Prozesses gewartet werden. Sie dürfen (ohne weitere Prüfung) davon ausgehen, dass die übergebenen Befehle eine maximale Länge von 100 (`CMD_MAX`) Zeichen nicht überschreiten. Zur Verwaltung der gestarteten Prozesse soll eine einfach verkettete Liste aus `struct process`-Elementen verwendet werden; das Hinzufügen von Elementen geschieht ebenfalls in der Funktion `main()`. Nach dem Start aller Befehle wird abschließend auf die noch laufenden Prozesses gewartet.
- Funktion `pid_t run(char *cmdline)`:
  Führt die übergebene Befehlszeile aus und gibt die Prozess-ID des erzeugten Kindes zurück. Dazu wird ein neuer Prozess erzeugt, das auszuführende Programm und die Parameter aus der Befehlszeile extrahiert und die Ausführung mittels einer Funktion der `exec()`-Familie gestartet. Tritt bei der Ausführung mittels `exec()` ein Fehler auf, wird eine aussagekräftige Fehlermeldung ausgegeben und der Kindprozess beendet. Zur Vereinfachung dürfen Sie annehmen, dass die zu extrahierenden Parameter durch Leerzeichen voneinander getrennt sind und sich innerhalb der einzelnen Parameter keine weiteren Leerzeichen befinden.
- Funktion `void waitProcess(void)`:
  Wartet passiv auf einen beliebigen der zuvor gestarteten Befehle. Nach Terminierung des Prozesses gibt `waitProcess` die PID, die Befehlszeile, den Exitcode (falls zutreffend) und die Ausführdauer aus. Nutzen Sie hierfür die Funktion `time()` (siehe Manpage).

Hinweise:

- Neue Prozesse sollen am Anfang der Prozessliste eingehängt werden.
- Die Definition der `struct process` darf um zusätzliche Einträge erweitert werden.
- Das `fflush()` am Ende des Programms ist in dieser Aufgabe nicht notwendig.
- Achten Sie auf **korrekte und vollständige Fehlerbehandlung**.

Auf den folgenden Seiten finden Sie ein grobes Gerüst für das beschriebene Programm. Es ist überall großzügig Platz gelassen, damit Sie nachträgliche Anpassungen durchführen können. Einige wichtige Manual-Seiten liegen bei – es kann aber durchaus sein, dass Sie nicht alle Funktionen oder gegebenenfalls auch weitere Funktionen benötigen.


```c
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <time.h>
#include <limits.h>
#include <errno.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>

#define CMD_MAX 100

static void die(const char *msg) {
  perror(msg);
  exit(EXIT_FAILURE);
}

static void usage(void) {
  fprintf(stderr, "Usage: hupsi <n>\n");
  exit(EXIT_FAILURE);
}

struct process {
  pid_t  pid; // PID des Prozesses
  // Eigene Mitglieder
};

// Makros, Funktionsdeklarationen, globale Variablen
```

G:

```
// Funktion main

  // Befehlszeilenargument(e) prüfen

  // Befehlszeilenarg. <n> mit strtol parsen

  // Befehle von stdin einlesen und verarbeiten
```



```
// Ende Funktion main
```

M:

```
// Funktion run
```

```
// Ende Funktion run
```

R:


```
// Funktion waitProcess
  // Auf beliebigen Prozess warten
  // struct process zu PID aus Prozessliste ermitteln
  // (Vorbereiten der) Ausgabe
```

```
// Prozess aus Liste entfernen & Speicher freigeben
```

```
// Ende Funktion waitProcess
```

W:

2) Makefile (8 Punkte)

Schreiben Sie ein Makefile, welches die Targets `all` und `clean` unterstützt. Ebenfalls soll ein Target `hupsi` unterstützt werden, welches das Programm `hupsi` baut. Greifen Sie dabei stets auf Zwischenprodukte (z.B. hupsi.o) zurück.

Das Target `clean` soll alle erzeugten Zwischenergebnisse und das Programm `hupsi` löschen.

Nutzen Sie dabei die Variablen `CC` und `CFLAGS` konventionskonform. Achten Sie darauf, dass das Makefile ohne eingebaute Regeln (Aufruf von `make -Rr`) funktioniert!

Mk:

Aufgabe 3: Synchronisation (14 Punkte)

1) Was versteht man unter einer Verklemmung und was sind die (hinreichenden und notwendigen) Bedingungen, damit eine Verklemmung auftreten kann? (6 Punkte)

2) Erläutern Sie das Konzept Semaphor. Welche Operationen sind auf Semaphoren definiert und was tun diese Operationen? (5 Punkte)