Theorie

1. Welchen Vorteil bietet die Behandlung von I/O über blockierende Prozesse und Interrupts gegenüber dem Polling-Verfahren?

2. Betrachten Sie den folgenden Programmausschnitt:

```
int pid1 = fork();
printf ("%s\n","[1] Ein Fork ist durch.");
if ( pid1==0 ) {
  printf ("%s\n","[2] Ich starte jetzt emacs.");
  execl ("/bin/emacs", "/etc/fstab", (char *)NULL);
  int pid2 = fork();
  printf ("%s\n","[3] Nach dem zweiten Fork.");
} else {
  int pid3 = fork();
  printf ("%s\n","[4] Da war noch ein Fork.");
};
printf ("%s\n","[5] Hier endet das Programm.");
```

Wie oft und warum erscheinen die mit [1] bis [5] durchnummerierten Ausgaben? Schreiben Sie zu jeder Ausgabe die Anzahl und begründen Sie Ihre Antwort stichwortartig – **oder** malen Sie eine Baumstruktur mit den Ausgaben auf.

3. Warum erhält bei `fork()` der Vaterprozess als Rückgabewert die PID des Sohnes, während der Sohn den Wert 0 erhält – warum kann es nicht andersrum sein (Vater erhält 0, Sohn erhält PID des Vaters)?

4. Während `fork()` einen Prozess (fast identisch) verdoppelt, müssen Sie bei der Thread-Erzeugung mit `pthread_create()` eine Funktion angeben, die im neuen Thread ausgeführt wird. Angenommen, es gäbe auch eine Funktion `thread_fork()` mit ähnlicher Semantik wie `fork()`, d. h., der neue Thread ist eine identische Kopie des ursprünglichen Threads. Die Funktion soll im aufrufenden Thread statt einer Prozess-ID die Thread-ID (in Linux-Sprache: die *Lightweight Process ID*) des erzeugten Threads zurückgeben, im erzeugten Thread 0 (also wie bei `fork`). Welches Problem ergäbe sich dann bei Code der Form

```
pid = fork_thread ();
if ( pid == 0 ) {
  /* ich bin der „Sohn“-Thread */
} else {
  /* ich bin der „Vater“-Thread */
};
```

Praxis

5. **ProcFS:** Das Proc-Dateisystem hält unter Linux für jeden Prozess (mit Prozess-ID *`PID`*) ein Unterverzeichnis `/proc/`*`PID`*`/` bereit, das zahlreiche (virtuelle) Dateien und Ordner enthält, in denen Sie Informationen über den jeweiligen Prozess finden.

a) Erzeugen Sie mit `touch /tmp/Z` eine Datei (`/tmp/Z`).
b) Über das Kommando `tail -f /tmp/Z &` (mit abschließendem „&“) lassen Sie sich als Hintergrundjob Änderungen an der Datei anzeigen.
c) Löschen Sie die Datei.
d) Betrachten Sie mit `ls -l /proc/$(pidof tail)/fd/` die Liste der von `tail` geöffneten Dateien. Was fällt Ihnen dabei auf?

6. **ProcFS:** In jedem der Proc-Ordner (`/proc/PID/`) ist `exe` ein Symlink auf die ausführbare Programmdatei, die in diesem Prozess läuft. Suchen Sie nur mit Hilfe dieser Information nach allen Prozessen, die `/bin/bash` (die Standard-Shell) ausführen.
7. **ProcFS:** Wenn Sie in einem Proc-Ordner die Datei `environ` ausgeben, sehen Sie das *Environment*, also die Liste aller Umgebungsvariablen, die für diesen Prozess gültig sind. Allerdings ist die Ausgabe unleserlich, weil Zeilenumbrüche fehlen. Probieren Sie stattdessen das folgende Kommando aus:

   ```
   tr '\0' '\n' < environ
   ```

   (hinter dem ersten Backslash steht eine Null, nicht der Buchstabe O) und finden Sie heraus, warum es funktioniert.