Übung 03

Betriebs- und Kommunikationssysteme
ProInf IV 2020
Simon Schmitt, Heiko Will

Abgabe freiwillig

Aufgabe 1: Prozesse & Threads

Beantworten Sie die folgenden Fragen, geben Sie ggf. Ihre Quellen an.

1. Was ist der **PCB**? Welche Einträge hat er und wozu dienen diese?
2. Grenzen Sie **Prozess** und **Thread** voneinander ab.
3. Warum ist eine einzige Warteschlange für Prozesse die auf I/O-Ereignisse warten in der Regel nicht sinnvoll?
4. Warum muss das Betriebssystem wissen, welche Dateien ein Prozess offen hat? Überlegen Sie sich zwei Situationen, in denen dieses Wissen nötig ist.
5. Schauen Sie sich den `stdio.h` (Standard Input/Output) Header der C-Bibliothek an. Nutzen Sie hierzu die Manpage (`man 3 stdio`) und verschaffen Sie sich einen Überblick über die zur Verfügung gestellten Funktionalitäten.
   a) Welche sind es (hier ist nur ein kurzer Satz gefordert, keine Auflistung der einzelnen Funktionen und ihrer Definitionen)?
   b) Grenzen Sie in diesem Zusammenhang die Begriffe **Stream** und **File Descriptor** voneinander ab.
   c) Wozu dienen `stdin`, `stdout` und `stderr`?

Aufgabe 2: `kill(1)`: Signale als Prozess-Interrupts

UNIX-oide Systeme stellen Syscalls bereit um sog. „Signale“ auszulösen und abzufangen. Man kann sich dies wie ein erweitertes Interrupt-Konzept vorstellen, mit dem Prozesse steuern können, was passieren soll, wenn im Rahmen ihrer Ausführung bestimmte Arten von Interrupts ausgelöst werden.

Ein Beispiel für ein solches Signal ist `SIGSEGV` oder auch „Segmentation Violation“. Der standardmäßige Handler für dieses Signal beendet das Programm und weist das System an, einen „coredump“ anzulegen und eine Fehlermeldung auf der Konsole auszugeben.

In dieser Aufgabe werden Sie sowohl ein Programm schreiben um Signale zu senden und eines um diese abzufangen:

1. Das erste Programm heißt „evil“ und soll das Betriebssystem anweisen, bei einem der Signale `SIGINT` oder `SIGTERM`, einen entsprechenden Handler („ISR“) auszulösen. Dieser Handler soll lediglich eine (böse) Nachricht ausgeben und das Programm *nicht* beenden.

   Hierzu nutzen Sie die Funktion `sigaction`, die eine Struktur `struct sigaction` erhält, welche beschreibt welche Funktion aufgerufen wird.

   Nachdem das Programm diese Routinen registriert hat, soll es mit der Funktion `getpid()` seine aktuelle PID herausfinden und ausgeben. Anschließend soll es in eine Endlosschleife verfallen und alle 2 Sekunden „Endlosschleife“ ausgeben. Nutzen Sie hierzu die Systemfunktion `nanosleep`.

   Testen Sie Ihr Programm, indem Sie nach dem Starten mit der Tastenkombination `Ctrl+C` das Signal `SIGINT` senden. Alternativ können Sie das Tool `kill` nutzen, um das Signal `SIGINT` mit `kill -2 eurePID` bzw. `SIGTERM` mit `kill -15 eurePID` an euer Programm zu senden. Beenden Sie das Programm mit `kill -9 eurePID`.

2. Nun schreiben Sie Ihre Version von `kill`. Sie bekommen zuerst optional eine Signalnummer (ansonsten implizit `SIGTERM`) übergeben und danach mindestens eine PID.

   Das Programm ruft für jede übergebene PID die Funktion `kill` mit dem Signal auf:

   ```
   $ kill 1234         # Sendet SIGTERM
   $ kill  -9 1234     # Sendet SIGKILL
   ```

   Testen Sie Ihr Programm `evil` nun mit Ihrer eigenen Version des Tools `kill`.

Achtung: Es ist nicht erlaubt in einem Signalhandler Funktionen wie `printf` zu verwenden. Nutzen Sie stattdessen die Systemschnittstelle `write`.

Hinweis: Sowohl für `sigaction`, `kill` als auch für `nanosleep` ist ein neuerer POSIX-Standard erforderlich als die GNU libc standardmäßig bereitstellt. Sie können mit

```
/* Use POSIX-Standard from 09-1993 */
#define _POSIX_C_SOURCE 199309L
```

vor den Includes in Ihrem Quellcode schreiben.

Da die vorgegebene Signatur des Signal-Handlers ein Argument vorsieht, das Sie aber nicht benötigen, müssen Sie eine entsprechende Warnung des Compilers unterdrücken:

```
void some_func(int unused_parameter) {
  (void)unused_parameter; // silence compiler
  /* ... */
}
```