

Systemmanagement und Sicherheit

6. Übung

Aufgabe 1 (fork-exec)

Entwickeln Sie ein C-Programm *start*, das beim Aufruf

```
start prog arg1 arg2 arg3 ...
```

zunächst ein *fork()* Aufruf ausführt und dann im Sohnprozeß das Programm *prog* via *execvp* mit den angegebenen Argumenten startet. Das Programm *prog* soll mit niedrigster Priorität ausgestattet werden, siehe Systemcall *setpriority()*

Der Vaterprozess soll weiterhin folgendes tun:

- Ausgabe der PID des gestarteten Prozesses *prog*,
- Ausgabe des Return-Codes von *prog* nach dessen Beendigung,
Hinweis: Siehe Macros unter *wait(2)*
- Ausgabe eines evtl. Signals (numerisch und eine Beschreibung des Signals),
das zum Abbruch von *prog* führte (siehe auch *psignal(3)*).

Die Deklaration von *main()* in *start.c* sei

```
int main(int argc, char **argv)
```

Hier ist ein richtiger Aufruf von *execvp()* dabei:

- a) `execvp(argv[1][0], argv[1])`
- b) `execvp(argc, argv)`
- c) `execvp(argv[1], argv[2])`
- d) `execvp(argv[1], argv+1)`
- e) `execvp(*argv[1], *argv[1])`
- f) `execvp(**argv, **argv[1])`
- g) `execvp(argv[1], argv[1])`

Aufgabe 2 (Semaphoren und Shared Memory)

Implementieren Sie folgendes Erzeuger–Verbraucher–Schema mit Hilfe von Shared Memory und Semaphoren. Ein Vaterprozess P_1 legt Semaphoren und Shared Memory Segment an. Danach erzeugt er den Sohnprozess P_2 , der als Prozesskopie die Semaphoren-ID und die Shared-Mem-ID kennt. Der Prozess P_1 wird Daten in den gemeinsamen Speicher schreiben, die Prozess P_2 dort herauslesen wird.

Der Erzeugerprozess P_1 hält ein Array gefüllt mit `int`-Daten, deren Anzahl sei durch eine

```
#define N_DATA 2000000
```

Direktive festgelegt. Die Daten werden von P_1 zufällig erzeugt (siehe `srand48()`, `lrand48()`).

Der Verbraucherprozess P_2 soll diese Daten erhalten, indem diese über einen von P_1 und P_2 genutzten shared memory Block übertragen werden. Im shared-memory Bereich finden weniger als `N_DATA` viele Zahlen Platz, etwa

```
#define N_SHARED 2000
```

Prozess P_1 muss also die größere Anzahl Daten in mehreren Durchläufen durch den kleineren Shared-Puffer übertragen.

Hinweise: es empfiehlt sich ein schrittweises Vorgehen

- zunächst eine Lösung ohne Semaphoren und nur einen Schreib-/Lesevorgang im Shared-Memory-Bereich:
 P_2 wartet mittels `sleep()`, damit P_1 Zeit hat, die Daten zu schreiben
- danach Semaphoren hinzunehmen
- danach mehrere Schreib-/Lesevorgänge

Aufgabe 3 (Signal-Handling `signal()`)

Schreiben Sie ein C-Programm `sigtest`, das einen einzigen Signalhandler für alle möglichen Signale besitzt. Der Signalhandler soll mittels `signal(3)` aktiviert werden.

Mittels `sleep(3)` soll `sigtest` für eine Minute existieren und innerhalb von main folgenden Return-Code zurückgeben:

- 0, falls kein Signal innerhalb dieser Minute empfangen, oder
- die Signalnummer

Die Kommunikation zwischen Signal-Handler und `main()`, um die Signalnummer mitzuteilen, soll über eine globale *volatile int* Variable `signo` realisiert werden.

Aufgabe 4 (Signal-Handling `sigaction()`)

Schreiben Sie analog zur vorherigen Aufgabe ein Programm *sigtest2*, das den gleichen Mechanismus mit Hilfe von `sigaction(2)` implementiert.

Aufgabe 5 (Signal-Handling (Signale künstlich erzeugen))

Benutzen Sie Ihr Programm *sigtest2* als Grundlage für ein Programm *sigtest3*, das mit einem erhöhten Parameter für `sleep()` arbeitet, um drei der in der Vorlesung angegebenen Signale zu simulieren. Hierbei bedeutet *Simulieren* nicht, dass mit Hilfe von `kill` oder `kill()` die jeweilige Signalnummer erzeugt wird. Vielmehr sollen beim Ablauf des Programms *sigtest* die Bedingungen erzeugt werden, die zum Senden des Signals führen, etwa dass ein Alarmtimer abläuft (siehe `alarm(3)`).