Praktikum Betriebssysteme – Übung 6

Aufgabe 1 – Verständnisfragen

a) Welche Mechanismen zur Prozesskommunikation kennen Sie?

a) Wodurch unterscheiden sich anonyme Pipes von benannten Pipes?

c) Bei der Kommunikation mittels Nachrichtenaustausch stehen verschiedene Arten der Adressierung zur Verfügung. Welche kennen Sie und welche Vor- bzw. Nachteile bieten diese?

Aufgabe 2 – Shared memory

a) Übersetzen Sie das Beispielprogramm für Shared Memory (→ git) und führen Sie es aus.

Das Programm erzeugt einen *anonymen shared-memory* Bereich und kopiert einen Textstring hinein. Anschließend wird `fork()` aufgerufen um einen Kindprozess zu erzeugen, der den Inhalt des gemeinsamen Speicherbereichs verändert. Der Elternprozess greift mehrfach auf den gemeinsamen Speicherbereich zu und gibt dessen Inhalt (vor und nach Veränderung durch den Kindprozess) aus.

b) Der Speicherbereich im Beispielprogramm ist nicht typisiert, wir schreiben mittels `memcpy()` einfach den Inhalt eines `char[]` hinein. Im Beispiel sind die beiden Texte, die in den shared-memory Bereich geschrieben werden nicht einmal gleich lang. Wieso funktioniert die Ausgabe des Textes mittels `printf()` trotzdem problemlos?

c) In der Regel würde man mit typisiertem Speicher arbeiten, also eine Datenstruktur im *shared-memory* Bereich anlegen. Modifizieren Sie das Programm so, dass im *shared-memory* Bereich eine kleine Datenstruktur (die z.B. fd-Nummer und Vorname enthält) abgelegt wird und modifizieren Sie die Daten abwechselnd im Eltern- und Kindprozess.

Hinweis: Auch die parallele Berechnung von Pi aus Übung 4.1 könnte bei der Verwendung von shared memory (oder einer anderen Kommunikationsform) mit Hilfe von Prozessen statt Threads gelöst werden.

Aufgabe 3 – Signale

a) Übersetzen Sie das Beispielprogramm für Signale (→ git) und führen Sie es aus.

Das Programm gibt seine Prozess-ID aus und ruft `sleep()` auf, um für eine Weile im Zustand *blockiert* zu bleiben. Senden Sie dem Programm von einem anderen Terminal aus verschiedene Signale. Dafür können Sie die Programme `kill` bzw. `pkill` verwenden. Vergleichen Sie das unterschiedliche Verhalten, falls die Standard-Aktion für das gesendete Signal „Terminate“, „Ignore“ oder „Stop“ ist.

Welche Signale existieren und welche Standard-Aktion für diese hinterlegt ist, erfahren Sie in der Manpage (`man signal.h`). Wie Sie Signale nutzen in der Manpage `sigaction` bzw. `signal`.

b) Überschreiben Sie den Signalhandler für das Signal SIGINT (Ctrl+C), so dass das Programm sich nicht mehr beendet, sondern z.B. einen Text ausgibt. Was können Sie beobachten, wenn Sie dem Programm nun aus einem anderen Terminal erneut das Signal SIGINT senden?

c) Sorgen Sie nun dafür, dass das Programm auch nach dem Empfang eines Signals weiterhin korrekt abläuft. Was ist dazu notwendig?

d) Wozu dient das Flag `SA_RESTART`? Funktioniert dieses um Aufgabe c zu lösen? Warum?

Aufgabe 4 – Anonyme Pipes

Anonyme Pipes haben Sie in der Shell bereits genutzt, wenn Sie zwei Kommandos mit dem `|`-Zeichen verbunden haben (z.B. `ps ax | grep kworker`). Das Beispielprogramm (→ git) zeigt Ihnen, wie Sie eine anonyme Pipe in einem C-Programm nutzen können. Auch die Manpages (`man 7 pipe`, `man 3 pipe`, `man 2 pipe`) haben eine Menge nützlicher Infos.

a) Übersetzen Sie das Beispielprogramm und führen Sie es aus. Das Programm erzeugt eine anonyme Pipe, ruft anschließend `fork()` auf um einen Kindprozess zu erstellen, und sendet eine Nachricht vom Elternprozess zum Kindprozess.

b) Erweitern Sie das Programm: Senden Sie eine Antwort vom Kindprozess zum Elternprozess.

c) Schreiben Sie ein Programm, das zwei andere Programme mittels anonymer Pipe verbindet. Die Ausgabe des einen Prozesses soll die Eingabe des anderen Prozesses werden. Auf diese Art können Sie ihre Shell-Implementierung um Pipes zu erweitern.

Der Aufruf kann zum Beispiel wie folgt aussehen: `./connect programm1 programm2`

Die Programme `programm1` und `programm2` sollen weiterhin einfach mittels `fork()` / `execvp()` aufgerufen werden, wie dies in der Shell-Implementierung bereits der Fall sein sollte. Sehen Sie sich in dem Zusammenhang die Manpage der Kommandos `pipe` und `dup2` an.

Zum Verständnis: Der Aufruf `./connect ps wc` soll die gleiche Ausgabe liefern, wie die Eingabe `ps | wc` in der Shell.

Aufgabe 5 – Benannte Pipes

Benannte Pipes (FIFOs) sind im Dateisystem als *special device* sichtbar. Sie können einerseits innerhalb ihres C-Programms erzeugt und genutzt werden, andererseits steht das Kommando `mkfifo` aber auch als Kommandozeilenprogramm zur Verfügung und erlaubt so die Nutzung benannter Pipes in der Shell. Selbstverständlich sind auch Mischungen möglich: Zum Beispiel ein C-Programm, das in die Pipe schreibt und ein Shell-Kommando (z.B. `cat`), das aus der Pipe liest.

a) Erzeugen Sie eine benannte Pipe auf der Kommandozeile. Sehen Sie sich anschließend die Datei im Dateisystem an (z.B. ls -al). Was fällt Ihnen auf?

b) Senden Sie eine Nachricht in die Pipe. Sie können das mit Hilfe der bereits bekannten Umleitung (>) tun, wobei hinter dem Umleitungszeichen der Speicherort Ihrer Pipe stehen muss. Wie verhält sich Ihre Eingabe auf der Kommandozeile?

c) Öffnen Sie ein zweites Terminal und lesen Sie in diesem aus der Pipe. Was passiert mit dem Programm im ersten Terminal wenn Sie im zweiten Terminal lesen? Können Sie den Effekt erklären?

d) Übersetzen Sie das Beispiel für benannte Pipes (→ git) und führen Sie es aus.

e) Schreiben Sie ein zweites Programm, das aus der Pipe lesen kann. Starten Sie dann sowohl das Programm aus Aufgabe d und das neue Programm, um eine Nachricht über die Pipe auszutauschen.