Aufgabe 2: Prozesssynchronisation

Ziel dieser Aufgabe ist es, den Umgang mit Prozesssynchronisation und Interprozesskommunikation zu üben.

ACHTUNG: Zu den untenstehenden Aufgaben existieren Vorgaben in Form von C-Dateien mit einem vorimplementierten Code-Rumpf, die ihr in den Aufgaben erweitern sollt. Diese Vorgaben sind von der Veranstaltungswebsite herunterzuladen, zu entpacken und zu vervollständigen! Die Datei vorgaben−A2.tar.gz lässt sich unter Linux/UNIX mittels `tar -xzvf vorgaben-A2.tar.gz` auspacken.

Primzahlserver (11 Punkte)

In der Vorgabe befinden sich sieben Dateien, die einen Primzahl-Server (der kontinuierlich Primzahlen *produziert*) und einen Client (der diese *konsumiert*) implementieren:

primserv.c Der Primzahl-Server, der in einer Endlosschleife Primzahlen produziert und in einer int-Variable in einem *Shared-Memory*-Segment (SHM-Segment) ablegt.

primeat.c Der Client, der ständig die int-Variable im *SHM*-Segment ausliest und diese ausgibt (also: Primzahlen aufisst!).

primshm.c, primshm.h Von Server und Client gemeinsam genutzter Code zum Anlegen und zur Verwaltung des *SHM*-Segments.

sync.c, sync.h (Fast) leere Dateien, in denen gemeinsam genutzter Code zur Prozesssynchronisation von euch abgelegt werden soll.

Makefile Eine Steuerdatei für GNU Make, mit dem ihr bequem die beiden Programme **primserv** und **primeat** erstellen könnt.

Make und Test der Vorgabe

1) make (1 Punkt) Erstellt die Programme **primserv** und **primeat**, indem ihr GNU Make (Kommando `make`) ohne weitere Parameter aufruft. Make gibt hierbei aus, welche Kommandos es genau ausführt – der Compiler wird je ein Mal zum Compilieren der beiden Programme aufgerufen. Verändert nun die Datei **primeat.c** und ruft erneut `make` auf. Welches Programm wird nun erneut compiliert? Warum ist das so? (⇒ antworten.txt)

2) Test (2 Punkte) Öffnet nun **zwei** Terminal-Fenster und startet im einen den vorher compilierten **primserv** (`./primserv`), im anderen den Primzahl-Esser **primeat** (`./primeat`). Was beobachtet ihr? Wieso „fehlen“ in den Ausgaben von **primeat** sehr viele der Primzahlen, die **primserv** erzeugt hat? (⇒ antworten.txt)

Synchronisation mit System-V-Semaphore

Das Produzieren (in primserv.c) und Konsumieren (in primeat.c) der Primzahlen soll nun synchronisiert werden, so dass keine Zahlen mehr verloren gehen. Das Szenario ist ein klassisches Erzeuger-/Verbraucher-Problem: Da der gemeinsam genutzte Puffer (die int-Variable im *SHM*-Segment, über die die produzierte Primzahl zu **primeat** gelangt) nur *ein* Element aufnehmen kann, reichen zur Synchronisation zwei Semaphore aus (wie in der Übung besprochen: jeweils eine zum Zählen der **noch freien** bzw. der **schon belegten** Elemente des Puffers).

a) Initialisierung, P und V (3 Punkte) Für die Semaphore-Implementierung sollen System-V-Semaphore zum Einsatz kommen. Um diese aus **primserv** und **primeat** leichter verwenden zu können, sollen zunächst die Funktionen im Modul **sync.c** (diese sind bereits in **sync.h** deklariert) mit Leben gefüllt werden:

- **int prim_sem_get()** soll mittels **semget(2)** eine Semaphore-Menge der Größe zwei anlegen, bzw. eine bereits existierende weiterverwenden, und die ID zurückliefern.
- **void prim_sem_init(int semid)** soll mittels **semctl(2)** (Kommando SETVAL) die beiden Elemente der Semaphore-Menge (Parameter semid) geeignet initialisieren. Diese Funktion soll später nur von **primserv**, nicht von **primeat**, aufgerufen werden.
- **void p(int semid, int sem_num)** (und analog: **v()**) soll durch Verwendung von **semop(2)** die Semaphore-Operation P (bzw. analog: V) implementieren, also das Dekrementieren (Inkrementieren) der Semaphore, die über die Parameter semid (ID der Semaphore-Menge) und sem_num (Nummer der Semaphore innerhalb der Menge) spezifiziert wird.

b) Verwendung in primserv und primeat (3 Punkte) Verwendet nun die in Aufgabenteil a) geschriebenen Funktionen in **primserv.c** und **primeat.c**. Ruft jeweils in der Funktion main() prim_sem_get() auf (in **primserv.c** zusätzlich prim_sem_init()) und sichert die zurückgegebene ID in einer globalen Variable. Synchronisiert das Erzeuger-/Verbraucher-Problem mit den zuvor geschriebenen Funktionen p() und v() und den beiden Semaphore.

Testet nun die so synchronisierte Fassung der Programme, wie bereits in Teilaufgabe 2) beschrieben.

3) Größerer Puffer (2 Punkte) Damit Erzeuger und Verbraucher stärker entkoppelt werden, wäre es wünschenswert, den verwendeten Puffer auf *mehrere Elemente* zu vergrößern. Beschreibt, welche Modifikationen an der gemeinsamen Datenstruktur (**primshm.h**, struct prim_shm) und der Synchronisation notwendig sind, um einen solche Puffer verwenden zu können. (⇒ antworten.txt)

c) Tatsächlich größerer Puffer (optional) Implementiert die in Aufgabenteil 3) skizzierten Änderungen und verwendet dafür einen Ringpuffer. Sorgt mittels Präprozessor (Stichwort „bedingte Übersetzung“) dafür, dass man leicht zwischen der „normalen“ einelementigen Implementierung und der Lösung zu Aufgabenteil c) wechseln kann, und kennzeichnet die veränderten Abschnitte explizit mit einem Kommentar.

Tipps zu den Programmieraufgaben:

- Einen eindeutigen Schlüssel für **semget(2)** kann man sich mit **ftok(3)** erzeugen.
- Werden Semaphore und gemeinsamer Speicher eines Prozesses nicht explizit freigegeben, bleiben diese auch über die Lebenszeit des Prozesses hinaus verfügbar. Mit dem Programm **ipcs(8)** könnt ihr euch die allokierten Semaphoren und den gemeinsamen Speicher anschauen, mit **ipcrm(8)** Löschungen vornehmen.
- Kommentiert euren Quellcode **ausführlich**, so dass wir auch bei Programmierfehlern im Zweifelsfall noch Punkte vergeben können!
- Die Programme sollen dem ANSI-C- und POSIX-Standard entsprechen und sich mit dem gcc auf den Linux-Rechnern im FBI-Pool übersetzen lassen. Das mitgelieferte Makefile ruft dazu den Compiler mit den Parametern -Wall -ansi -pedantic -D_XOPEN_SOURCE -D_POSIX_SOURCE auf. Falls ihr die Programme dagegen in C++ schreiben möchtet, muss in der ersten Zeile des Makefile „gcc“ mit „g++“ ersetzt werden.

Abgabe: bis spätestens Dienstag 08:00 in der Woche nach der Tafelübung, d.h. 26. Mai (Übungsteilnehmer 1./3./5./... Woche) bzw. 02. Juni (2./4./6./... Woche)