BIS15	VZ	Betriebssysteme und Systemprogrammierung	
Übungsblatt	5	Thema: IPC-Ressource-Nutzung	WS 2016/17

Beispiel zu IPC-Ressource-Nutzung
Entwickeln Sie die 5 Versionen (A, B, C, D und E) dieser Übung mit **POSIX** IPC Funktionen.
Für die Version B kopieren sie einfach den Ordner der Version A und nehmen anschließend die Änderungen im neuen Ordner vor.

Die Namen für die benötigten Queues, des Shared Memory bzw. der Semaphoren sollen mit der Matrikelnummer beginnen.
(nach dem Schema: /is1510xx_foo , foo = Platzhalter für beliebiges Wort)

A)
Entwickeln Sie 2 eigenständige Programme „Menu“ und „Display“, die als eigenständige Prozesse laufen. Prozess Menu soll dem Prozess Display Nachrichten mithilfe einer Message Queue senden.

Menu:

Liest in einer Schleife Daten (von Standard Input) ein und schreibt Zeile für Zeile in die Message Queue, bei “quit” wird beendet

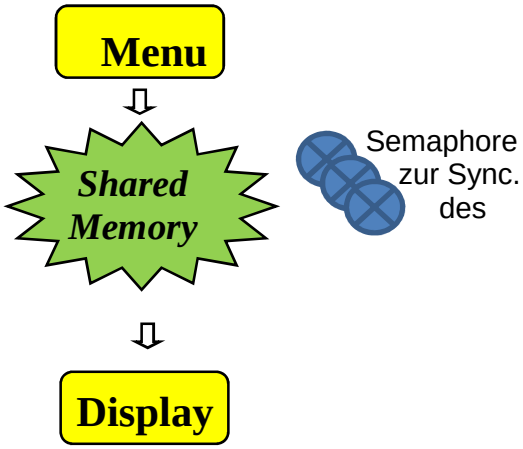
Display:

Liest aus der Message Queue und gibt jede Zeile auf dem Bildschirm aus (Standard Output).

Achten Sie besonders darauf, dass nach der Eingabe von **quit** bei Menu auch der Prozess Display beendet wird und am Ende keine Message Queue mehr existiert!

B)
Ersetzen Sie aus der Version A, die aus den Prozessen Menu und Display besteht, die Message Queue durch einen Shared Memory.

Ändern Sie das Beispiel so ab, dass der Prozess Menu im Shared Memory die Zeile platziert. Display liest die Zeile aus dem Shared Memory. Sorgen Sie mit geeigneten Semaphoren dafür, dass das Display erst liest, wenn Menu eine Nachricht bzw. Zeile im Shared Memory platziert hat und dass umgekehrt Menu erst eine neue Nachricht bzw. Zeile ins Shared Memory schreiben kann, wenn Display die Zeile vom Shared Memory gelesen hat.



Es ist NICHT verlangt, dass der Shared Memory mehrere Zeilen zwischenspeichern kann. Sondern es soll immer genau eine Zeile von Menu in den Shared Memory gespeichert werden. Anschließend liest Display die Zeile aus und Menu darf die nächste Zeile speichern usw.

Relevante Funktionen für Shared Memory sind:
shm_open(), ftruncate(), mmap(), munmap(), close(), shm_unlink().

Relevante Funktionen für Semaphore sind:
sem_open(), sem_wait(), sem_post(), sem_close(), sem_unlink()

Achten Sie besonders darauf, dass nach Eingabe von **quit** bei Menu auch der Display Prozess beendet wird und am Ende kein Shared Memory und keine Semaphoren mehr existieren!

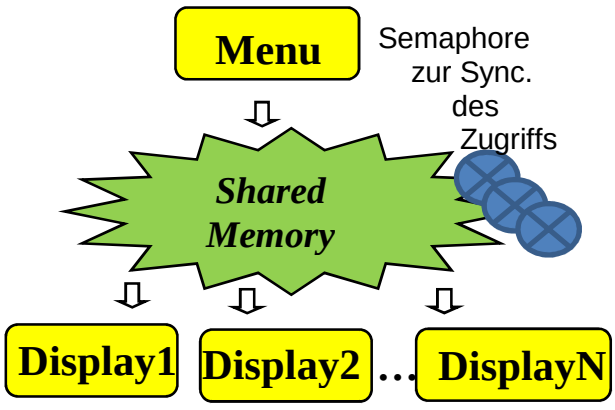
C)
Erweitern Sie Version B um weitere Display Prozesse. Menu schreibt in den Shared Memory und alle gestarteten Displays lesen anschließend die Nachricht vom Shared Memory und geben sie aus. Erst wenn alle Display Prozesse die Nachricht gelesen haben, darf die nächste Nachricht von Menu geschrieben werden.

Die einzelnen Prozesse Display1, Display2 bis DisplayN sollen dasselbe Binary benutzen (Also es soll display nicht doppelt programmiert/kopiert werden für jede Display Instanz,
sondern in einem C-File). Falls die Anzahl der Displays und der jeweilige Display-Index für euch (bzw. euren Quellcode) relevant sind, könnt ihr euch die Anzahl und den Index als Argument übergeben.

```
/home/alice/ue4b $ ./menu 4      # startet Menu und uebergibt Anzahl der Displays
/home/alice/ue4b $ ./disp 0      # startet Display 1
/home/alice/ue4b $ ./disp 1      # startet Display 2
/home/alice/ue4b $ ./disp 2      # startet Display 3
/home/alice/ue4b $ ./disp 3      # startet Display 4
```

Falls ihr eine (elegantere) Lösung ohne zusätzliche Angabe von Argumenten bei Menu und Display entwickelt, können die Args beim Starten entsprechend weggelassen werden.

Achten Sie besonders darauf, dass nach Eingabe von **quit** bei Menu auch die Display Prozesse beendet werden und am Ende kein Shared Memory und keine Semaphoren mehr existieren!
Überlegen Sie sich, ob es in dieser Version C einen Vorteil bringt, wenn der Menu Prozess sich als aller letzter Prozess beendet. (Beide Varianten sind möglich)

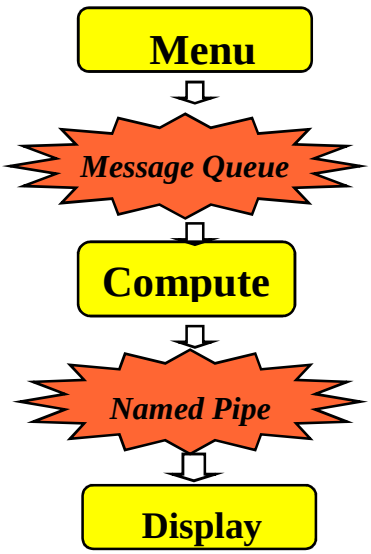


D)
Erweitern Sie die Version A, die aus den Prozessen Menu und Display besteht um einen Compute Prozess (eigenes Programm mit eig. C-File).
Entwickeln Sie 3 eigenständige Prozesse, die folgendermaßen Kommunizieren:

Menu:
Liest in einer Schleife Daten (von Standard Input) ein und schreibt Zeile für Zeile in die Message Queue, bei “quit” wird beendet

Compute:
Liest Zeile für Zeile aus der Message Queue, wandelt das Gelesene in Großbuchstaben um und schreibt das in die Named Pipe

Display:
Liest aus der Named Pipe und gibt jede Zeile auf dem Bildschirm aus



Die Named Pipe soll im aktuellen Verzeichnis angelegt werden und als Namen die Matrikelnummer is1510xx verwenden. Der Display Prozess darf die Nachricht von Compute erst ausgeben, wenn die ganze Nachricht empfangen wurde. Das bedeutet der Display Prozess muss in der Lage sein, das Ende einer Nachricht und den Beginn einer neuen Nachricht in der Pipe zu erkennen. (z.B: durch einen Zeilenumbruch)

Achten Sie besonders darauf, dass nach Eingabe von **quit** bei Menu auch der Compute und der Display Prozess beendet werden und am Ende die Message Queue und die Named Pipe nicht mehr existieren!

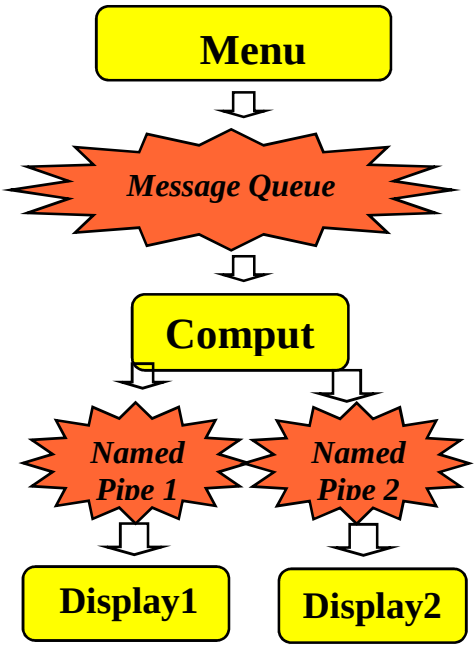
Relevante Funktionen für Named Pipe (FIFO) sind:
mkfifo() zum Erstellen einer Named Pipe und die Funktionen für die „normalen“ Dateizugriffe (entweder per POSIX Syscalls (read, write, …) oder per gebufferte Stream IO Funktionen (fwrite, fread, …).

E)
Erweitern Sie Version D um einen weiteren Display Prozess. Das bedeutet es gibt einen Menu Prozess, einen Compute und zwei Display Prozesse.

Menu:
Liest in einer Schleife Daten (von Standard Input) ein und schreibt Zeile für Zeile in die Message Queue, bei “quit” wird beendet

Compute:
Liest Zeile für Zeile aus der Message Queue, wandelt das Gelesene einmal in Großbuchstaben um und einmal in Kleinbuchstaben um. Die großgeschriebene Nachricht kommt in die Named Pipe1 und die kleingeschriebene in die Named Pipe2. (z.B: „Hallo“ → „HALLO“ in Pipe1 und „hallo“ in Pipe2

Display:
Liest aus der richtigen Named Pipe und gibt jede Zeile auf dem Bildschirm aus.



Die Prozesse Display1 und Display2 sollen dasselbe Binary benutzen (Also es soll Display nicht doppelt programmiert/kopiert werden, sondern in einem C-File) und über ein Argument beim Starten erfahren die Display Prozesse, ob sie Display1 oder Display2 sind und können entsprechend die richtige Named Pipe öffnen.

Vorbereitung für Kernel Programmierung:

Richten Sie Ihren Linux Rechner zur Programmierung von Kernel Modulen ein. Kompilieren Sie das „Hello World“ – Kernel-Modul vom eCampus.
Kompilieren mit „make“. Anschließend sollten sie das Kernel Modul cdrv.ko im Verzeichnis haben. Wenn die Kompilierung erfolgreich ist, haben sie auf ihrem System alle notwendigen Packages installiert. Das „Hello World“ Kernel Modul soll nicht am eCampus abgegeben werden, sondern nur das IPC Beispiel mit den Versionen A bis E.