Aufgabe 7:

buffer-Modul (12 Punkte)

Programmieren Sie (in Zweiergruppen) ein Modul **`jbuffer.c`**, das einen Datenstrom von einem Filedeskriptor einliest, eine gewisse P1ufferung vornimmt und ihn dann auf einem anderen Filedeskriptor etwas zeitversetzt wieder ausgibt. Solche Mechanismen werden beispielsweise eingesetzt, um Schwankungen in der Ankunftsrate von Audiodaten (Jitter) auszugleichen und eine gleichmäßige Wiedergabe zu ermöglichen.

Ihr Modul soll eine Funktion mit folgender Schnittstelle bereitstellen:

```
void jbuffer(int fd1, int fd2, int bufsize, int bufdelay)
```

Die Funktion sorgt dafür, dass ein Bytestrom von Filedeskriptor **`fd1`** in einen Puffer der Größe **`bufsize`** ingelesen wird und die Daten auf dem Filedeskriptor **`fd2`** ausgegeben werden, sobald der Puffer erstmals den Füllstand **`bufdelay`** erreicht hat.

Das Modul soll mit zwei Threads (Posix-Threads) arbeiten. Der erste Thread nimmt die Initialisierung vor (Puffer anlegen, zweiten Thread starten, etc.) und liest dann die Daten von **`fd1`** und schreibt sie in den Puffer. Der zweite Thread wartet bis der Füllstand **`bufdelay`** erreicht ist und beginnt dann mit der Ausgabe auf **`fd2`**. Der Puffer soll als Ringpuffer betrieben werden. Zur Synchronisation der beiden Threads beim Puffer-Zugriff (Anfangs-Synchronisation, voller Puffer, leerer Puffer, Endsituation) eignen sich am besten zählende Semaphoren, für einfachen gegenseitigen Ausschluss auf gemeinsame Daten reichen ggf. auch Mutex-Variablen. Sie können statt mit Semaphoren aber auch direkt mit Mutex- und Condition-Variablen koordinieren.

Nachdem auf **`fd1`** End-of-File erreicht wurde gibt Thread2 noch alle im Puffer befindlichen Daten aus, danach gibt das Modul alle belegten Ressourcen frei, Thread2 terminiert und die Funktion **`jbuffer`** kehrt zurück.

Wenn Sie mit zählenden Semaphoren koordinieren wollen, müssen Sie diese selbst auf der Basis von Mutex- und Condition-Variablen realisieren. Implementieren Sie dazu am besten ein Modul **`mysem.c`** mit den Funktionen:

`int *sem_init(int n)` legt eine neue Semaphore an, initialisiert sie mit dem Wert **`n`** und liefert einen Zeiger darauf zurück.

`void P(int *s, int n)` erniedrigt Semaphore **`s`** um **`n`**, blockiert wenn nötig

`void V(int *s, int n)` erhöht Semaphore **`s`** um **`n`**, weckt blockierte Threads auf

`void sem_delete(int *s)` gibt Semaphore s frei.

Erstellen Sie ein zu der Aufgabe passendes Makefile.

Beschreiben Sie in einer Datei **jbuffer.txt** welche Synchronisationssituationen in der Aufgabe auftreten und wie Sie diese jeweils mit welchen Hilfsmitteln synchronisieren.

Hinweise:
In **`/proj/i4sos/pub/aufgabe7/jbuftest.c`** finden Sie ein Testprogramm, zu dem Sie Ihre Implementierung binden können. Dieses Programm liest Daten von der Datei **`/proj/i4sos/pub/aufgabe7/input`** und schreibt auf die lokale Datei **`output`**. Überprüfen Sie mit **diff(1)**, dass die beiden Dateien identisch sind.

Die pthread-Funktionen sind in einer speziallen Funktionsbibliothek zusammengefasst, die Sie beim ompilieren bzw. Binden Ihres Programms mit einbinden müssen. Geben Sie hierzu nach den .c- bzw. .o-Dateien die Option **`-lpthread`** an.

Abgabe: bis spätestens Montag, 05.07.2004, 10:00 Uhr