Aufgabe 8:

23.06.2005

buffer-Modul (12 Punkte)

Programmieren Sie (in Zweiergruppen) ein Modul **`jbuffer.c`**, das einen Datenstrom von einem Filedeskriptor einliest, eine gewisse Pufferung 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`** eingelesen wird und die Daten auf dem Filedeskriptor **`fd2`** ausgegeben werden, sobald der Puffer erstmals den Füllstand **`bufdelay`** erreicht hat.

Die Funktion **`jbuffer`** soll nach der Initialisierung (Puffer anlegen, etc.) hierzu zwei Threads (Posix-Threads) erzeugen, die die Aufgabe nebenläufig erledigen und nach deren Terminierung zurückkehren.

Der erste Thread liest die Daten zeichenweise von **`fd1`** (am besten einfach mit getc(3)) und schreibt sie in den Puffer. Der zweite Thread wartet zunächst bis der Füllstand **`bufdelay`** erreicht ist und beginnt dann mit der Ausgabe auf **`fd2`** (am besten mit putc(3)). 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 Semaphore, 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.

Thread 1 terminiert nachdem auf **`fd1`** End-of-File erreicht wurde. Thread2 gibt danach noch alle im Puffer befindlichen Daten aus und terminiert dann ebenfalls. Die Funktion **`jbuffer`** gibt nach dem Ende beider Threads alle belegten Ressourcen frei und kehrt zurück.

Wenn Sie mit zählenden Semaphoren koordinieren wollen, (vgl. auch Vorl., Folien VIII-30 ff.) müssen Sie diese selbst auf der Basis von Mutex- und Condition-Variablen realisieren. Implementieren Sie dazu am besten ein Modul **`mysem.c`** mit folgenden Funktionen (**`SEM`** ist dabei eine zu definierende Datenstruktur (typdef!), die den Semaphor-Zustand sowie die intern benötigten Mutex- und Condition-Variablen enthält):

`SEM *sem_init(int n)`	legt einen neuen Semaphor an, initialisiert ihn mit dem Wert **`n`** und liefert einen Zeiger darauf zurück.
`void P(SEM *s, int n)`	erniedrigt Semaphor **`s`** um **`n`**, blockiert wenn nötig
`void V(SEM *s, int n)`	erhöht Semaphor **`s`** um **`n`**, weckt blockierte Threads auf
`void sem_delete(SEM *s)`	gibt Semaphor 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/aufgabe8/jbuftest.c`** finden Sie ein Testprogramm, zu dem Sie Ihre Implementierung binden können.
Dieses Programm liest Daten von der Datei **`/proj/i4sos/pub/aufgabe8/input`** und schreibt auf die 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 Compilieren bzw. Binden Ihres Programms mit einbinden müssen (Compiler-Option **`-lpthread`**).

Abgabe: bis spätestens Dienstag, 05.07.2005, 17:30 Uhr