Hochschule RheinMain
Fachbereich DCSM - Informatik
Prof. Dr. Robert Kaiser
Sebastian Flothow
Alexander Schönborn
Daniel Schultz

Betriebssysteme

WS 2020/21
LV 3122

Aufgabenblatt 5
Bearbeitungszeit 2 Wochen
Abgabetermin: 08.02.2021, 4:00 Uhr

In dieser Aufgabe werden gemeinsam genutzte Speichersegmente (Shared Memory Segment oder Shared Memory Object, kurz *SHM*, vgl. Kap. 8.7) betrachtet. Als Programmierschnittstelle soll das POSIX-API benutzt werden:

`int shm_open(const char *name, int oflag, mode_t mode);`	Öffnen oder Erzeugen und Öffnen eines POSIX SHM mit einer initialen Größe von 0 Bytes.
`int ftruncate(int fd, off_t length);`	Setzen der Größe des Shared Memory Segments.
`int shm_unlink(const char *name);`	Entfernen eines Shared Memory Segments.
`void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);`	Einblenden einer Datei oder eines Shared Memory Objektes in den Adressraum des aktuellen Prozesses.
`int munmap(void *addr, size_t length);`	„Ausblenden“ eines Teils des Adressraum.

Auf Linux-Systemen sind die POSIX Shared Memory Segmente im Dateisystem unter `/dev/shm` sichtbar. Weiterführende Informationen finden Sie in der Manual-Page `shm_overview`.

Achtung: MacOS unterstützt keine Unnamed Semaphoren. Bitte lösen Sie diese Aufgabe auf einem Linux-System!

Aufgabe 5.1 (Leser und Schreiber):

In dieser Aufgabe wird ein Shared Memory Segment als einelementiger Kommunikationspuffer zur Realisierung einer Kommunikationsbeziehung zwischen mehreren Schreiber-Prozessen und einem Leser-Prozess verwendet. Die Schreiber-Prozesse legen Nachrichten ab, der Leser-Prozess liest auf Anforderung durch den Benutzer den Kommunikationspuffer und gibt die aktuell enthaltene Nachricht auf dem Bildschirm (über Standardausgabe) aus. Das Shared Memory Segment ist damit ein gemeinsam benutztes Betriebsmittel aller Prozesse.

a) Schreiben Sie ein Programm `create.c`, das ein Shared Memory Segment mit einem über die Kommandozeile übergebenen Namen und einer festen Größe erzeugt, die für die im Folgenden betrachtete Kommunikationsaufgabe ausreichend ist. Die Struktur des Shared Memory Segments soll in `shared.h` beschrieben werden.

b) Schreiben Sie ein Programm `destroy.c`, welches ein mit dem Programm aus (a) erzeugtes Shared Memory Segment löscht. Auch hier soll der Name der Shared Memory Segments über die Kommandozeile angegeben werden.

c) Schreiben Sie ein Programm `schreiber.c`, dem beim Aufruf über `argv[]` der Name des Shared Memory Segments, ein Zeichen `x` (z.B. 'A'), eine Länge `length` (≤ 40) und ein Wartezeit-Parameter `sleep_time` (in Millisekunden, z.B. 100) übergeben werden. Das Programm mache sich das in (a) erzeugte Segment zugänglich. Es schreibe dann zyklisch 10 mal eine Nachricht in das Shared Memory Segment mit folgender Struktur: `(int pid, int i, int length, xx...x)`. (Achtung: Es sind die Datentypen *binär* abzulegen und nicht in Zeichenketten zu wandeln!) Dabei sei `pid` die Prozess-ID des ausführenden Prozesses, `i` die Nummer des aktuellen Iterationsschritts, `length` (s.o.) die mit dem Aufruf übergebene Anzahl der folgenden Zeichen `xx...x`, wobei `x` für das mit dem Aufruf übergebene Zeichen steht (hierdurch können Sie die Ausgaben von verschiedenen Prozessen einfach unterscheiden). Achtung: die Zeichenfolge `xx..x` bildet keinen Null-terminierten String, deshalb muss die Anzahl der abgelegten Zeichen über `length` explizit mitangegeben werden. Nach der Ausgabe eines Zeichens sowie (zusätzlich) nach Abschluss jeder Iteration lege sich der Prozess für `sleep_time` Millisekunden schlafen, bevor er das nächste Zeichen schreibt bzw. mit dem nächsten Iterationsschritt beginnt (zeitliche Dehnung, damit Sie besser beobachten können).

d) Schreiben Sie ein Programm `leser.c`, das sich zu Beginn das Shared Memory Segment zugänglich macht (Name über Kommandozeile) und anschließend bei Eingabe eines Zeichens von der Standardeingabe mittels `getchar()` die Datenstruktur im Shared Memory Segment liest und als lesbare Zeile z.B. in der Form `pid, i: xx...x` ausgibt.

Hinweis: Testen Sie die Programme mit dem Shell-Skript `run.sh`, das im Hintergrund mehrere parallele Schreiber-Prozesse und im Vordergrund einen Leser-Prozess startet.

Aufgabe 5.2 (Synchronisierung):

In dieser Aufgabe sollen die Ausgaben der Schreiber-Prozesse synchronisiert werden (konsistente Ausgabe *eines* Iterationsschrittes). Legen Sie dazu zusätzlich eine POSIX-Semaphore mit im Shared-Memory Segment an, welche die Schreiber untereinander synchronisiert.

a) Erweitern Sie das Programm `create.c` um die Initialisierung des Semaphors. Beachten Sie, dass auf das Semaphor-Objekt von mehreren Prozessen gleichzeitig zugegriffen werden soll (`pshared = 1`).

b) Erweitern Sie `schreiber.c` entsprechend um die geforderte Synchronisierung.

Aufgabe 5.3 (Speisende Philosophen):

In dieser Aufgabe wird das Problem der speisenden Philosophen betrachtet. Gehen Sie von der korrekten Lösung des Philosophenproblems aus, wie sie in der Vorlesung (Folien 5-71ff) für N Philosophen vorgestellt wurde. Die Philosophen werden auf UNIX-Prozesse abgebildet.

Programmieren Sie die Lösung, indem Sie die gemeinsam benutzten Daten aller Philosophen sowie die benötigten Semaphoren in einem Shared Memory Segment anlegen. Der initiale Prozess soll hierbei alle gemeinsamen IPC-Objekte erzeugen. IPC-Objekte sind Mittel, welche dazu benötigt werden, die kontrollierte gemeinsame Verwendung von Betriebsmitteln zwischen Prozessen zu ermöglichen. Hierzu zählen unter Anderem Anonyme Shared Memory Segmente (siehe

`MAP_ANONYMOUS` bei `mmap()`) und Semaphoren aus der bekannten POSIX API. Weiterhin soll der Prozess für jeden Philosophen jeweils einen Kindprozess erzeugen, wobei jeder Kindprozess die `philosopher()`-Funktion ausführt. Eine Überlagerung mittels `execve()` wird nicht gefordert.

Bestimmen Sie zunächst die benötigte Anzahl der Semaphoren. Denken Sie anschließend über deren Bedeutung sowie über sinnvolle Initialwerte für diese nach. Die Kindprozesse sollen insgesamt 10 mal speisen, bevor sie terminieren. Der Elternprozess wartet, bis alle Kindprozesse fertig sind, und gibt danach die benutzten IPC-Objekte wieder frei.

Organisieren Sie die Ausgabe des Programms so, dass die relevanten Zustände der Philosophen sichtbar werden und alle Ausgaben eines Philosophen untereinander in einer Gruppe von Spalten erscheinen. Testen Sie die Lösung für N=1, 2, 3, 5 und 8 Philosophen.

Hinweise:

- Für den Wert `i = 0` liefert das Makro `LEFT (i-1)%N` aus den Vorlesungsfolien das Ergebnis -1 und nicht $N-1$. Benutzen Sie stattdessen folgende Makros:

```
#define N         5
#define LEFT(x)   (((x)-1+N)%N)
#define RIGHT(x)  (((x)+1)%N)
```

- Prüfen Sie mit `assert()`, ob sich die Philosophen tatsächlich in den korrekten Zuständen befinden.