Systemprogrammierung: Client und Server mit Shared Memory

Klaus Kusche

Unsere **`vmstat`**-Programme aus den ersten Übungen haben den Fehler,
dass man die Systemauslastung nur ab dem Start des Programms sieht,
aber nicht für die Vergangenheit.

Wir wollen das mit einem Client-Server-Programm lösen:

- Der *Server läuft ständig* (*ohne* Ausgabe) und *speichert* den Output von **`vmstat`**
  in einem Shared Memory (*zyklisch* für die letzten *n* Sekunden).
- Der *Client* gibt die *im Shared Memory gespeicherten Werte* grafisch aus.
  Damit man die Ausgabe zeitlich zuordnen kann, soll jeder Eintrag
  im Shared Memory neben den drei CPU-Werten einen *Zeitstempel* enthalten.

Da diesmal *unabhängig voneinander gestartete Programme* (also nicht durch **`fork`**
aus einem einzigen Vater entstandene Prozesse) auf Shared Memory und Semaphore
zugreifen sollen, brauchen wir *die benannte Variante von Shared Memory und Semaphore*,
nicht die anonyme.

a) Header-File

Die **`.c`**-Files von Client und Server inkludieren beide einen *gemeinsamen Header*,
der vor allem das Speicher-Layout des Shared Memory festlegt. Er enthält:

- Ein **`#include`** für **`time.h`** (für **`time_t`**, den Typ der *Zeitstempel*)
- *Namen* für das Shared Memory und die Semaphore
  (irgendein "Filename" mit **`/`** am Anfang):
  Client und Server müssen beide *dieselben Namen* verwenden,
  um auf dasselbe Shared Memory und dieselbe Semaphore zuzugreifen
- Eine *Konstante* für die *Anzahl der Einträge* im Shared Memory
- Einen *Strukturtyp* für *einen* Eintrag:
  - Ein **`time_t`** für den *Zeitstempel*
  - Drei **`short int`**'s (aus Platzgründen) für **`us`**, **`sy`** und **`wa`**
- Einen *Strukturtyp* für das *ganze* Shared Memory:
  - Ein **`int`** für den *Index des ältesten Eintrags* im Array
    (weil wir das Array ja zyklisch beschreiben)
  - Ein *Array* von Einträgen mit der oben festgelegten Größe

Wer das noch kann, macht ein schönes **`#ifndef`** um den Header!

b) Server

Teile des Servers lassen sich aus unserem **`popen`**-**`vmstat`**-Programm übernehmen.

- Da der Client ***sofort mit Fehlermeldung enden*** soll, wenn der Server nicht läuft,
  schreiben wir eine *Aufräum-Funktion*. Sie macht (ohne Fehlerprüfung!)
  - ein **`munmap`** und ein **`shm_unlink`** für das *Shared Memory*
  - ein **`sem_close`** und ein **`sem_unlink`** für die *Semaphore*

- ein **pclose** auf die *Pipe* vom **vmstat**
- ein **exit**

Dafür müssen der **FILE*** der Pipe und die *Pointer* auf Shared Memory und Semaphore *global* sein.

Die Aufräum-Funktion wird *überall* dort aufgerufen, *wo das Programm endet* (also z.B. nach allen Fehlermeldungen statt **exit**). Außerdem schreiben wir einen *Signal Handler*, der auch *nur diese Aufräum-Funktion aufruft*.

Das Server-**main** enthält folgende Schritte:

- *Anlegen des Shared Memory* mit **shm_open**, **ftruncate** und **mmap**.

  Das Shared Memory soll *read/write* angebunden werden und für andere Programme die Rechte **0644** haben.

  Im Server soll *auf jeden Fall ein neues Shared Memory* angelegt werden: Wenn es schon eines mit diesem Namen gibt, soll der Server mit *Fehlermeldung* enden.

  Nach **mmap** kannst du den File-Deskriptor von **shm_open** gleich wieder mit einem normalen **close** *schließen*.

  Der Inhalt des Shared Memory muss nicht explizit initialisiert werden, es wird automatisch komplett mit **0** gefüllt.

- *Anlegen der Semaphore* mit **sem_open**:

  Auch auf jeden Fall *eine neue*, keine bestehende, mit Rechten **0666** und Zählerstand **1**.

- *Einrichten unseres Signal Handlers* für **SIGINT**, **SIGTERM** und **SIGHUP**.

- *Starten* des **vmstat** mit **popen** (wie in der alten Übung).

- *Zeilenweises Lesen* des **vmstat**-Outputs aus der Pipe:
  - Die ersten drei Zeilen werden wieder *überlesen*.
  - Das gesamte Ändern des Shared Memories *innerhalb* eines Schleifendurchlaufes wird *durch die Semaphore geschützt*.
  - *In den bisher ältesten Eintrag* im Shared Memory wird die *aktuelle Zeit* gespeichert (du bekommst sie von **time(NULL)**).
  - Die gelesene Zeile wird wie bisher mit **sscanf** verarbeitet, die *drei CPU-Werte* werden direkt in den bisher ältesten Shared-Memory-Eintrag gespeichert. **Achtung:** Das **scanf**-Format für **short int** ist **%hd** (*"half decimal"*)!
  - Der *Index des ältesten Eintrags* im Shared Memory wird um 1 erhöht, und zwar *zyklisch* (d.h. *modulo Arraygröße*).

- Falls die Schleife endet (was sie nicht tun sollte): **pclose** mit Auswertung der Todesursache des **vmstat**.

c) Client

Es ist in den meisten Fällen sehr unklug, eine Semaphore gesperrt zu halten, während man *I/O-Funktionen oder andere potentiell langdauernde Operationen* aufruft: Würden wir während der gesamten Ausgabe des Shared Memories die Semaphore gesperrt halten, und würde der Benutzer den Output des Clients in ein **less** pipen

und minutenlang betrachten, könnte der Server das Shared Memory minutenlang *nicht aktualisieren*.

Da es aber schwierig ist, im Client nur pro Eintrag zu sperren und auf dazugekommene bzw. weggefallene Einträge richtig zu reagieren, wählen wir einen anderen Ansatz:
Wir legen im Client eine *globale Variable mit derselben Struktur wie das Shared Memory* an, *kopieren* das Shared Memory in einem Rutsch (gesperrt) in diese Variable,
und geben dann deren Inhalt aus. Das *minimiert die Sperrzeit* auch wenn die Ausgabe blockiert und garantiert uns einen konsistenten Snapshot der Daten.

Das **`main`** des Clients besteht aus folgenden Schritten:

- *Anbinden der Semaphore* (**`sem_open`**), diesmal muss sie *schon existieren*
  (warum ist es klug, das *vor* dem Shared Memory zu machen,
  also im Vergleich zum Server *in umgekehrter Reihenfolge*?)
- *Anbinden des Shared Memory* (**`shm_open`** und **`mmap`**),
  ebenfalls nur dann, wenn es schon existiert, und *nur zum Lesen*.
- *Kopieren des Shared Memory* in die globale Variable mit **`memcpy`**,
  geschützt durch die Semaphore.
- **`munmap`** des Shared Memory, **`close`** seines Filedeskriptors und **`sem_close`**
  der Semaphore (*kein* **`unlink`**, denn beide solle ja nach dem Client *weiterleben*!).
- *Ausgabe aller Einträge aus der Kopie* des Shared Memory:
  - Die Schleife *beginnt beim ältesten Eintrag*.
  - Es werden nur die Einträge ausgegeben, deren *Timestamp nicht 0* ist
    (in Einträgen mit Timestamp 0 wurde noch nie etwas gespeichert).
  - Vorne in jeder Zeile wird der *Zeitstempel* ausgegeben
    (ich habe den Zeitstempel mit **`ctime`** in einen String verwandelt,
    das **`\n`** am Ende dieses Strings durch ein Leerzeichen *ersetzt*,
    und diesen String ausgegeben.
    Du kannst aber auch mit **`localtime`** und **`printf`** arbeiten).
  - Dann werden die *drei Balken* ausgegeben
    (*Hilfsfunktion* aus der alten Übung verwenden).
  - Am Ende wird der Schleifenzähler (Position im Array) *zyklisch erhöht*.
    Kommt man wieder beim ältesten Eintrag an, *endet* die Schleife.

Hinweise:

- Bring das Programm *zuerst ohne Semaphore* zum Laufen
  und schreib erst dann den Code für die Semaphore dazu.
- *Prüfe* alle zentralen System- und Library-Aufrufe auf Fehler
  und gib "schöne" *Fehlermeldungen* aus (mit Programmname, **`errno`**-Text usw.).
- Du benötigst beim Linken **`-lpthread`** und **`-lrt`** .

Experimente:

- Zur Überprüfung:
  - Du solltest dein Shared Memory und deine Semaphore unter **`/dev/shm`** sehen.
  - Nach *Ende des Servers* sollten sie *dort wieder weg* sein,
    und der *Aufruf des Clients* sollte mit einer entsprechenden *Fehlermeldung* enden.