Systemprogrammierung: Shared Memory & Race Conditions

Klaus Kusche

Wir wollen ein Programm schreiben, in dem eine *wählbare Anzahl von Söhnen*
parallel jeweils *mehrere Daten-Einträge in einem Shared Memory* ablegen.
Nach dem Ende der Söhne soll der Vater prüfen, ob das fehlerfrei gelungen ist.

Unser Shared Memory enthält ein *Array von Strukturen*.
Jedes Element enthält die *Nummer des Sohnes*, von dem es stammt (von 1 aufwärts),
und eine *Nummer*, die angibt, um den *wievielten Eintrag dieses Sohnes* es sich handelt
(auch von 1 aufwärts).

Im Detail:

- Deklariere zuerst den oben erwähnten *Strukturtyp* mit zwei **`int`**'s.
- Prüfe und verarbeite dann die *Befehlszeile*.
  Das Programm soll mit *zwei positiven* **`int`**'s aufgerufen werden:
  Das *Anzahl der Sohn-Prozesse* und der *Anzahl der Einträge*,
  die jeder Sohn im Array speichern soll
  (wie verwandelt man ein numerisches Befehlszeilen-Wort in einen **`int`**?).
- Dann kommt das *Anlegen das Shared Memory*:
  Es wird im Vater eingerichtet und beim **`fork`** an die Söhne weitergegeben.
  Für diese Zwecke reicht ein *anonymes* **`mmap`** (d.h. ein **`mmap`**,
  das *nicht* mit einem File oder einem Shared-Memory-Namen verbunden ist).
  Es hat auch den Vorteil, dass es beim Anlegen automatisch auf 0 initialisiert wird.

  Unser **`mmap`** hat:
  - Keine fix festgelegte Adresse im Speicher
  - Platz für ((Anzahl der Söhne) * (Anzahl der Einträge) + 1) viele Strukturen
    (wie ermittelst du den Platz, den du pro Struktur brauchst?)
  - Lese- und Schreibrechte
  - Typ *anonym und geshared* (sonst bekäme jeder Sohn eine eigene Kopie!)
  - Und daher keinen File (**`-1`**) und keine Anfangsposition im File

  Überlege:
  - Wie gibt man in einem **`mmap`**-Parameter *zwei oder mehr Flags kombiniert* an?
    (die Flags sind numerische Konstanten, in denen je ein Bit gesetzt ist)
  - *Welchen Typ* nimmst du für die Variable, die das *Ergebnis* des **`mmap`** speichert?
    (im Hinblick darauf, dass wir in unserem Shared Memory dann
    ein Array von Strukturen speichern wollen)
- Dann kommt eine *Schleife*, die die angegebene *Anzahl von Söhnen erzeugt*
  (d.h. ein **`fork`** enthält).
  - Der *Vater* macht in der Schleife nur das **`fork`**,
    d.h. dann gleich den nächsten Schleifendurchlauf.
  - Der *Sohn* enthält eine *Schleife* mit so vielen Durchläufen
    wie Elemente im Array zu speichern sind.

    Für jedes Element *sucht* der Sohn zuerst mittels Schleife den *ersten freien Platz*

im Shared-Memory-Array (zu erkennen daran, dass die *Sohn-Nummer eines freien Eintrags* **0** ist; deine **fork**-Schleife zählt hoffentlich ab **1**, nicht ab **0** ?). Beginne die Suche nicht jedesmal wieder bei Eintrag 0, sondern an der Stelle, an der der Sohn seinen *vorigen Eintrag* gespeichert hat!

In diesen freien Eintrag *speichert* der Sohn *seine Sohn-Nummer* und seine *fortlaufende Eintrags-Nummer*.

Nach dem Speichern aller seiner Einträge *endet* (!) der Sohn-Prozess.

- Dann kommt nochmals eine *Schleife*, die so oft **wait** aufruft, wie es Söhne gibt.
- Schließlich prüft der Vater, *wie viele Einträge tatsächlich* im Shared Memory gespeichert wurden (d.h. sucht mittels Schleife den ersten freien Eintrag), und gibt die *Sollzahl* (Anzahl Söhne * Anzahl Einträge pro Sohn) und die *Ist-Zahl* aus.

  Optional kann der Vater auch noch *alle Einträge* der Reihe nach ausgeben (oder wenn du es ganz luxuriös machen willst: Die *fehlenden* und die *doppelten* Einträge anzeigen!).

Hinweise:

- *Prüfe* alle zentralen System- und Library-Aufrufe auf Fehler und gib "schöne" *Fehlermeldungen* aus (mit Programmname, **errno**-Text usw.).
- **MAP_ANONYMOUS** steht nicht im Standard. Falls du den Compiler so aufrufst, dass er nur Standard-konforme Programme akzeptiert, musst du vor allen **#include** ein **#define _GNU_SOURCE** schreiben.

Experimente:

- *Ab welcher Anzahl* von Einträgen kommt es zu *Fehlern*?

  Ein Sohn-Prozess sollte natürlich *immer* fehlerfrei funktionieren.
  Bei mir funktioniert alles unter insgesamt 1000 Einträgen relativ zuverlässig (also z.B. 10 Prozesse mit je 100 Einträgen oder 100 Prozesse mit je 10 Einträgen).
  Ab 1000 Einträgen treten manchmal Fehler auf, ab 100000 Einträgen (also z.B. 100 Prozesse mit je 1000 Einträgen) treten immer und massiv Fehler auf.

- Laufen deine Söhne *echt parallel*?

  Kommentiere dazu die zeitfressende Ausgabe nach Ende der Söhne komplett aus und miss dann die *Programmlaufzeit* mit dem Befehl **time**.
  Wie kannst du aus der Ausgabe von **time** auf parallel laufende Söhen schließen?

Erweiterung: Semaphore

Wir wollen das Problem fehlender und doppelter Einträge lösen, indem wir unsere Shared-Memory-Zugriffe durch eine Semaphore absichern.
Wir verwenden dazu eine *namenlose POSIX-Semaphore*:
Da die Semaphore so wie das Shared Memory im Vater angelegt und einfach von allen Söhnen geerbt wird, braucht sie keinen Namen.

- Damit alle Söhne ein und dieselbe Semaphore verwenden (und nicht jeweils eine eigene Kopie davon!), kann die Semaphore allerdings *nicht* als normale **sem_t**-Variable im Vater angelegt werden, sondern muss ebenfalls in einem Shared Memory liegen.

Man könnte dazu unser bestehendes **`mmap`** vergrößern und sie dort hineinlegen.
Ich habe aber ein *zweites* **`mmap`** gemacht (so groß wie eine **`sem_t`**-Variable, sonst genauso wie das erste).
Überlege: *Mit welchem Typ* deklarierst du den Pointer auf das zweite **`mmap`**?

- Als nächstes gehört die Semaphore mit **`sem_init`** *initialisiert*
(noch *vor* der **`fork`**-Schleife),
und zwar als *"shared Semaphore"* (gemeinsam für mehrere Prozesse)
mit Zählerstand **`1`** (weil ja maximal 1 Prozess die kritische Region betreten soll).
- Zuletzt gehört ein **`sem_wait`** und ein **`sem_post`** rund um die kritische Region *im Sohn*-Code.

  Überlege: *Was gehört zur kritischen Region*, was nicht?
  Du sollst das Shared Memory *nicht für die gesamte Laufzeit* (Anlegen aller Einträge) eines Sohnes monopolisieren!

Hinweise:

- Die Funktionen für POSIX Semaphore sind unter Linux nicht in der normalen C-Library, sondern in der pthread-Library.
  Du musst daher mit **`-lpthread`** compilieren bzw. linken.

Experimente:

- *Stimmt die Gesamtzahl* der Einträge jetzt immer? (sie sollte!)
- Wie wirkt sich die Semaphore auf die *Laufzeit* aus? Auf die *Parallelität*?

Erweiterung: Atomic Compare-And-Swap

Die Verwendung einer Semaphore löst das Problem zwar, aber beeinträchtigt die Performance massiv. Wir wollen daher prüfen, ob sich das Problem stattdessen auch mit einer *atomaren Operation* lösen lässt. Voraussetzung für eine "einfache" Lösung mittels atomarem Befehl ist, dass die *kritische Operation* im Wesentlichen nur aus einem *Lesen und Schreiben einer einzelnen Variable* besteht.

Wir haben Glück: Es gibt mehrere Möglichkeiten,
das *Reservieren eines Eintrags im Array atomar* durchzuführen:

- Eine Idee wäre, im Shared Memory zusätzlich den *Index des ersten freien Elementes* zu speichern (oder einen Pointer auf das erste freie Element).
  Dann könnte das Reservieren mit einem *"atomic Increment"* dieses Index erfolgen.

  Diese Idee ist sogar *effizienter* als das Suchen des ersten freien Platzes mittels Schleife und wäre zu bevorzugen, aber wir wollen unser Shared Memory strukturell nicht ändern, also brauchen wir eine andere Idee.
- Die zweite Idee ist, das Reservieren selbst, also das *Speichern der Sohn-Nummer* im Element, atomar mit der *"frei"-Prüfung* zu koppeln: "*Speichere nur dann die eigene Sohn-Nummer im ersten freien Eintrag, wenn da noch 0 drinsteht*".

  Gelingt das, haben wir den Eintrag erfolgreich reserviert und speichern auch die fortlaufende Nummer im Eintrag. Ist die Sohn-Nummer aber *nicht mehr* 0, hat uns inzwischen jemand diesen freien Eintrag weggeschnappt:
  Wir müssen *nochmals* das (inzwischen andere) erste freie Element *suchen*

und wieder versuchen, es atomar zu belegen.

Die zweite Idee wollen wir umsetzen.
Gehe dazu von der ursprünglichen Lösung (*ohne* Semaphore) aus.

- Die dazu nötige Operation *"prüfe, ob eine Variable einen bestimmten Wert enthält, und nur wenn ja, speichere einen neuen Wert"* heißt in der Informatik ***CAS: "Compare and Swap"***.
  Die x86-Prozessoren haben dafür einen eigenen Maschinenbefehl **CMPXCHG** (*"Compare and Exchange"*). Der **gcc**-Compiler bietet dafür eine *Builtin-Funktion*
  `bool __sync_bool_compare_and_swap(int *ptr, int oldval, int newval)`

  Ein Aufruf dieser Funktion wird vom Compiler durch einen **CMPXCHG**-Befehl mit **LOCK**-Prefix (exklusiver Speicherzugriff) ersetzt.
  `ptr` zeigt auf die *Speicherstelle*, die geprüft und geändert werden soll,
  `oldval` ist der *bisherige Wert*, auf den geprüft wird,
  und **`newval`** ist der *neue, zu speichernde Wert*.
  Das Ergebnis ist **`true`** bei Erfolg und **`false`** wenn **`*ptr`** *nicht* **`oldval`** enthält.
- Das bisherige *Speichern der Sohn-Nummer* im ersten freien Array-Element wird durch einen *Aufruf dieser Funktion* ersetzt.
- Dieser Aufruf steht in der *Bedingung* einer **`do`**-**`while`**-*Schleife*, die *rund um die Such-Schleife für das erste freie Element* steht (denn wenn das CAS *schiefgeht*, müssen wir nochmals den ersten freien Eintrag im Array suchen).

Experimente:

- *Stimmt die Gesamtzahl* der Einträge auch wieder bei jedem Versuch? (sie sollte!)
- Sind *Laufzeit* und *Parallelität* besser als bei der Semaphore-Lösung?