POSIX/C Multithreading: Mutex, Bedingungsvariablen, Erzeuger/Verbraucherproblem

Aufgabe 1: Mutex

Übersetzen Sie das gegebene Programm **`no01Vorgabe.c`** (mit `ring.h`) und starten Sie es.

In `ring.h` ist eine Warteschlange (queue) als „Ring-Array“ implementiert. Ein Schreibindex verweist auf die nächste Schreibposition, ein Leseindex auf die nächste Leseposition. Bei jeder Schreib- oder Leseoperation (enqueue, dequeue) wird der entsprechende Indexverweis weiter bewegt, am Ende des Arrays wird wieder vorn begonnen. Die Queue ist leer, wenn Schreib- und Leseindex gleich sind. Sie ist voll, wenn der Schreibindex genau eine Position hinter dem Leseindex steht. Um *voll* und *leer* unterscheiden zu können, bleibt ein Platz immer ungenutzt.

Ein Erzeuger-thread schreibt im 100 ms Takt fortlaufend die Zeichen 'A' bis 'Z' in eine Warteschlange. Ein Verbraucher entnimmt alle 10 ms ein Zeichen und gibt es auf `stdout` aus. Nicht belegte Plätze in der Schlange werden mit '-' dargestellt.

- Studieren Sie den Code, um die Funktionsweise zu verstehen.
- Variieren Sie die Frequenzen des Erzeugers und Verbrauchers (im `main`). Tauschen Sie insbesondere die Verzögerungszeiten.
- Variieren Sie die Kapazität der Warteschlange in `ring.h` (jetzt 50).

Da keine Synchronisation der threads eingebaut ist und der kritische Bereich *Queue* in keiner Weise abgesichert ist, funktioniert das Programm nur fehlerhaft:

- Sobald der Zustand *voll* erreicht wird, ist die Reihenfolge der Zeichen falsch.
- Der Verbraucher entnimmt auch die '-' Zeichen.

Korrigieren Sie nun das Programm wie folgt:

- Sichern Sie den kritischen Bereich mit einem **Mutex**. Da er zu dem Ring gehört, deklarieren Sie ihn in der `RingTy`-Struktur und initialisieren Sie ihn in `getNewRing()`.
  Vermutlich haben sich Ihre threads nun verklemmt (deadlock), da `enqueue` oder `dequeue` mit `return` den kritischen Bereich verlassen hat, ohne den Mutex freizugeben.
- Die beiden Zugriffsoperationen dürfen beim Zustand *voll* bzw. *leer* nicht einfach zurückkehren. Sie müssen warten, bis sich der Zustand ändert.
  Prüfen Sie daher die voll/leer-Bedingungen jeweils in einer Schleife (`if` in `while` ändern).
  Innerhalb der Schleifen darf natürlich kein `return` ausgeführt werden.
  Eine Zustandsänderung kann immer nur von dem jeweils anderen thread herbeigeführt werden (ein „Puffer-voll-enqueue“ wartet auf dequeue oder ein „Puffer-leer-dequeue“ wartet auf enqueue). Daher muss man in den beiden `while`-Schleifen dem jeweils anderen thread die Möglichkeit bieten, den Mutex zu belegen:
  -> ein *unlock Mutex* in den Schleifen ausführen

  Gleich im nächsten Schleifendurchlauf ist der voll/leer-Test jetzt ungeschützt, die Anweisungen nach der Schleife sind auch ungeschützt und es wird am Ende ein unlock ausgeführt, ohne vorher den Mutex zu belegen. Daher:
  -> zusätzlich gleich ein *lock Mutex* nach dem *unlock* in den Schleifen ausführen

Das Programm sollte nun korrekt arbeiten:

- Es werden nur fortlaufend die Zeichen A bis Z ausgegeben, keine '-' Zeichen mehr.
- Die Geschwindigkeit der Ausgabe wird vom langsamsten thread bestimmt, da beide gegebenenfalls aufeinander warten. Auch delay=0 für beide threads ist möglich.

Beobachten Sie die CPU-Auslastung mit der grafischen Anzeige der Systemüberwachung. Offenbar wird der Rechner von dem Programm stark beschäftigt, da die Zustände voll/leer bei dieser Lösung durch ständige Abfragen (*polling*, *busy-wait*) geprüft werden.

Aufgabe 2: **Condition Variables**

Reduzieren Sie die CPU-Auslastung des Programms durch die Verwendung von zwei Bedingungsvariablen.
Die Deklarationen können Sie auch der `RingTy`-Stuktur zufügen, die Initialisierung erfolgt wieder in `getNewRing()`.

Mit diesen beiden Variablen können die threads sich gegenseitig mitteilen, dass eine Zustandsänderung eingetreten ist. Dieses Signalisieren erfolgt jeweils nach der Änderung des Pufferinhalts. In den Schleifen wird jetzt auf eine Bedingungsvariable gewartet.

Beobachten Sie wieder die CPU-Auslastung. Da die threads nicht mehr aktiv warten, sollte eine deutlich geringere Auslastung erkennbar sein.

Wie ist die Auslastung, wenn beide delays=0 oder beide delays=1ms sind?

Aufgabe 3: **Andere Ausgabe**

Es sollen jetzt nicht nur die mit `dequeue` abgeholten Zeichen angezeigt werden, sondern immer der gesamte Inhalt der Queue - inklusive des unbesetzten Elements:

- in der `verbrauche`-thread-Funktion nicht mehr den `dequeue`-Wert ausgeben (der `dequeue`-Aufruf muss aber bleiben!)
- im `main` ständig (z.B. alle 50ms) den gesamten Pufferinhalt in einer Zeile ausgeben (`fflush(...)` nicht vergessen, Ausgabepuffer leeren)

Wenn Sie die Geschwindigkeiten variieren können Sie jetzt beobachten, wie sich die Schlange füllt oder entleert. Mindestens ein '-' Zeichen muss immer zu sehen sein.

Da jetzt ein dritter thread auf den kritischen Bereich zugreift, können Fehler auftreten. Um dies zu zeigen, verzögern Sie die Ausgabe in der inneren Schleife um z.B. 10ms.
Korrigieren Sie nun das Fehlverhalten:

-> auch die `main`-Ausgabe-Schleife mit dem Mutex schützen

Aufgabe 4: **voll ... leer ... voll ... leer ...**

Um das Auffüllen und Entleeren besser beobachten zu können, tauschen Sie die delays des Erzeugers und Verbrauchers jedesmal, wenn der Zustand *voll* oder *leer* erreicht wird.

Die Queue sollte sich jetzt ständig füllen, leeren, füllen, ...

Wer die Ausgabe immer in derselben Zeile haben möchte, gibt am Ende statt `'\n'` nur `'\r'` aus.