2 Synchronisation und Thread Generierung

Im **ersten Teil** dieser Aufgabe wird das Erzeuger Verbraucher Problem mit zwei unterschiedlichen Varianten zur Synchronisation der Threads realisiert.

- **Variante 1** verwendet nur die Synchronisationselemente Mutex und Semaphore.
- **Variante 2** verwendet nur die Synchronisationselemente Mutex und Conditional Variable.

Im **zweiten Teil** der Aufgabe werden Thread Pools für die Erzeuger und Verbraucher verwendet.

Die gesamte Aufgabe muss in C implementiert werden. In dieser Aufgabe müssen mehrere C Module übersetzt und gelinkt werden. Sie werden unterschiedliche Implementierungsvarianten über `#define` Variablen steuern. Dieser Compilations- und Konfigurationsprozess soll über ein **Makefile** gesteuert werden.

Hinweis: Diese Aufgabe hat mehrere Schwierigkeitsgrade. Zum einen müssen Sie unterschiedliche Synchronisationskonzepte durchdringen und umsetzen. Weiterhin üben Sie den Umgang mit `make`. Schließlich muss der Code auch noch gut strukturiert sein.
Folgende Vorgehensweise kann Ihnen die Arbeit erleichtern:

- Erstellen Sie ein Konzept für Teil 1. Lassen Sie sich dabei Zeit, diskutieren Sie Alternativen in der Gruppe. Arbeiten Sie vorab die benötigten Techniken durch.
- Nehmen Sie das Makefile aus Aufgabe 1 als Basis für das hier zu schreibende Makefile. Erstellen Sie das Makefile am Anfang und ergänzen es bei Bedarf. So können Sie die Vorteile von `make` während der Erstellung der Aufgabe nutzen.
- Keine Angst vor Refactoring, auch wenn es mal eine Viertelstunde dauert. Sie sparen in Summe Zeit und durchdringen die Thematik besser.
- Gehen Sie anschließend genauso mit Teil 2 der Aufgabe vor.

Abgabe des Konzepts vor dem Praktikumstermin

- Erstellen Sie jeweils ein Konzept für die beiden Teilaufgaben.
- Nutzen Sie bei der Erstellung des Konzepts Notationen und Techniken, die sie bisher im Studium gelernt haben.
- Schicken Sie das Konzept via Mail am Abend vor Ihrem Praktikumstermin am Malte Nogalski und Franz Korf.

Teil 1

Das Erzeuger - Verbraucher System besteht aus folgenden Threads:

- **Main Thread:** Dieser Thread führt alle notwendigen Initialisierungen durch und erzeugt alle weiteren Threads des Systems. Anschließend geht er in den Zustand blocked über und wartet, bis die von ihm erzeugten Threads terminiert sind. Danach beendet er das Programm.

- **Producer_1 Thread:** Dieser Thread erzeugt alle 3 Sekunden ein Zeichen, das er in einen FIFO Puffer schreibt. Er nimmt zyklisch jeweils das nächste Zeichen aus dem String “abc ... xyz“. Ist der Puffer voll, blockiert er bis wieder Speicherplatz im Puffer frei ist. Während dieser Wartezeit ist er im Zustand “blocked“. Producer_1 gibt auf dem Bildschirm aus, dass er ein Zeichen in den Puffer geschrieben hat.
- **Producer_2 Thread:** Dieser Thread ist analog zum Producer_1 Thread aufgebaut. Im Unterschied zu Producer_1 schreibt er Großbuchstaben in den Puffer.
- **Consumer Thread:** Dieser Thread nimmt alle 2 Sekunden ein Zeichen aus dem FIFO Puffer und gibt es auf dem Bildschirm aus. Ist der Puffer leer, wartet er im Zustand “blocked“, bis ein neues Zeichen im Puffer liegt.
- **Control Thread:** Dieser Thread steuert das Erzeuger - Verbraucher System auf Basis von Tastatureingaben wie folgt:
  - Tastatureingabe 1: Starte / Stoppe Producer_1
  - Tastatureingabe 2: Starte / Stoppe Producer_2
  - Tastatureingabe c oder C: Starte / Stoppe Consumer
  - Tastatureingabe q oder Q: Termination des Systems
  - Tastatureingabe h: Liefert diese Liste von möglichen Eingaben
  - Alle anderen Tastatureingaben werden ignoriert.

Implementation des FIFOs: Implementieren Sie ein eigenes C Modul für den FIFO. Synchronisieren Sie den Zugriff auf den FIFO Puffer über einen Mutex.
Der Puffer soll maximal 10 Elemente speichern. Verwenden Sie in **Variante 1** Semaphoren als Füllstandsanzeiger, so dass kein Overflow bzw. Underflow auftritt. Realisieren Sie dies in **Variante 2** über Conditional Variablen.

Hinweis: Beachten Sie bei der **Termination** des Systems, dass die Erzeuger bzw. Verbraucher Threads aufgrund eines vollen bzw. leeren FIFOs blockiert sein können. **Es ist nicht zulässig, dass weitere Elemente produziert bzw. konsumiert werden, nur weil ein Verbraucher bzw. Erzeuger aufgrund des Füllstands des Puffers blockiert ist.** Das bedeutet, dass die Threads vermutlich über `pthread_cancel` beendet werden müssen - s. Vorlesung: Unterschiedliche Techniken zur Termination eines Threads.
Sie müssen sicherstellen, dass ein Thread nicht innerhalb eines kritischen Abschnitts (critical section) beendet wird. Natürlich ist es erlaubt und notwendig, dass ein Thread beendet wird, während er auf das Betreten eines kritischen Abschnitts wartet. Schauen Sie sich dazu die Dokumentation der entsprechenden Funktionen bezüglich der Aspekte an, ob sie einen Cancellation Point besitzen und wie sie auf `pthread_cancel` reagieren. Achten Sie auf die Einstellung der Zustandsvariablen `cancelstate` und `canceltype`.

Anforderungen an die Lösung

- Die Synchronisation muss so realisiert sein, dass ein Thread, der auf ein Ereignis wartet, im Zustand “blocked“ ist, d.h. Polling ist nicht erlaubt.
- Ein Erzeuger - oder Verbraucher Thread blockiert, wenn der Control Thread ihn anhält, d.h. auch hier ist Polling nicht gestattet. Dieses Verhalten kann man zum Beispiel mit einem binären Semaphore realisieren.

- Erzeuger bzw. Verbraucher können aufgrund eines vollen bzw. leeren Puffers blockieren. In diesem Fall muss das System auch "sauber" beendet werden. Insbesondere dürfen keine zusätzlichen Elemente erzeugt bzw. verbraucht werden, damit Erzeuger bzw. Verbraucher nicht weiter blockieren.
- Schreiben Sie ein Makefile, dass über "make all" die C Dateien compiliert und bindet. Der Aufruf "make clean" löscht das ausführbare Programm und alle temporären Dateien.
- Geben Sie angemessene Fehlermeldungen aus. Greifen Sie dazu auf die Fehlerrückgabewerte der Funktionen zurück, die über errno bzw. gemäß POSIX Standard bereitgestellt werden.
- **Strukturieren Sie das Programm so, dass Codesequenzen nicht kopiert werden.** Kapseln Sie z.B. die Synchronisationsvarianten (s. Variante 1 und 2), so dass sie problemlos austauschbar sind. Bevor Sie Code kopieren, führen Sie bitte ein Refactoring durch.
- Gehen Sie bitte im Konzept auf die von Ihnen verwendeten Synchronisationselemente und deren Einsatz in Ihrer Lösung ein.
- Nachdem die Threads beendet wurden müssen die Ressourcen der verwendeten Mutexe und Semaphore über die entsprechenden destroy Funktionen dem BS zurück gegeben werden. Hier müssen Sie beachten, dass zum Beispiel die destroy Funktion für einen gelockten Mutex fehlschlägt.

Teil 2

Hier werden zwei Thread Pools mit jeweils N (= 5) Threads realisiert. Der erste Pool führt Erzeuger - Tasks aus, der zweite Pool Verbraucher - Tasks.
Für jedes Zeichen, das erzeugt werden soll, wird eine Task in die Task Queue des entsprechenden Pools gelegt. Die Task erhält das Zeichen als Parameter und schreibt nur dieses Zeichen in den FIFO. Dann schläft sie 3 Sekunden und ist anschließend beendet. Analog wird für jedes Zeichen, das aus dem FIFO gelesen werden soll, eine Task in die Task Queue des Pools für die Verbraucher gelegt. Nachdem das Zeichen ausgelesen und ausgegeben wurde, schläft sie 2 Sekunden und ist anschließend beendet.

Bezüglich der Synchronisation von Erzeuger und Verbraucher sollen auch hier die beiden Varianten aus Teil 1 unterstützt werden. Bei einer entsprechenden Strukturierung des Programms ist dies problemlos und ohne Aufwand möglich.

Damit besteht in diesem Teil der Aufgabe das System aus folgenden Threads:

- **Main Thread:** Dieser Thread führt alle notwendigen Initialisierungen durch, erzeugt pro Pool eine Auftragsqueue und die Worker Threads der Pools. **Unter Emil finden Sie ein Modul für solch eine Queue.**
  Weiterhin erzeugt der Main Thread die drei folgenden Threads. Anschließend geht er in den Zustand blocked über und wartet, bis die von ihm erzeugten Threads terminiert sind. Danach beendet er das Programm.
  - **Producer Generator Thread:** Dieser Thread erzeugt jede Sekunde eine Produce Task.

- **Consumer Generator Thread:** Dieser Thread erzeugt jede Sekunde eine Consume Task.
- **Control Thread:** Dieser Thread steuert das System wie folgt:
  - Tastatureingabe p oder P: Starte / Stoppe den Producer Generator Thread.
  - Tastatureingabe c oder C: Starte / Stoppe den Consumer Generator Thread.
  - Tastatureingabe q oder Q: Termination des Systems.
  - Tastatureingabe h: Liefert diese Liste von möglichen Eingaben
  - Alle anderen Tastatureingaben werden ignoriert.

Anforderungen an die Lösung

- Natürlich müssen auch hier die Anforderungen aus Teil 1 erfüllt werden. Da Sie den Code aus Teil 1 hier wiederverwenden, sollte dies kein Problem sein.
- Ergänzen Sie das Makefile aus Teil 1 entsprechend.
- Die Work Queues sollen jeweils 50 Aufträge speichern können. Die maximale Größe von Message Queues , auf denen die Work Queue basiert, kann im `proc` Filesystem über den Befehl
  `sudo sh -c "echo 200 > /proc/sys/fs/mqueue/msg_max"`
  angepasst werden.