Technische Universität Berlin Fachgebiet Komplexe und Verteilte IT-Systeme Sommersemester 2024	**Praxisaufgabe 2** zu – Systemprogrammierung – Prof. Dr. Odej Kao
Abgabetermin:	**12.07.2024 23:59 Uhr**

In dieser Hausaufgabe sollen Sie einen thread-safen Ringbuffer für Nachrichten variabler Größe implementieren.

Aufgabe 1: Threadloser Ringbuffer (12P)

Sie sollen einen Ringbuffer für Nachrichten variabler Größe erstellen, der zunächst ohne Thread-Sicherheit implementiert wird. Dadurch können Sie die grundlegende Logik und Funktionalität des Ringbuffers testen und sicherstellen. Im Einzelnen bedeutet dies: Sie können zunächst davon ausgehen, dass keine zwei Prozesse gleichzeitig auf den Ringbuffer zugreifen (weder lesend noch schreibend).

Verwenden Sie die vorgegebenen Datenstrukturen und Funktionen aus der Datei *include/ringbuf.h* und ergänzen Sie den Code an den markierten Stellen in der Datei *src/ringbuf.c*.

Funktionsweise des Ringbuffers

Die Informationen über den Ringbuffer werden in einer Variable des Typs *rbctx_t* gespeichert. In diesem struct sind folgende Elemente enthalten:

- `uint8_t * begin` Anfang des Speichers des Ringbuffers
- `uint8_t * end` Adresse einen Schritt nach dem letzten nutzbaren Byte im Ringbuffer
- `uint8_t * read` Position des Read-Pointers
- `uint8_t * write` Position des Write-Pointers
- `pthread_mutex_t mtx` Mutex-Variable, die von den Threads genutzt werden soll um gleichzeitiges Lesen bzw. Schreiben zu vermeiden
- `pthread_cond_t sig` Signal, auf das Threads warten

Am Anfang befinden sich sowohl Read- als auch Write-Pointer ganz am Anfang des Ringbuffers (siehe Abbildung 1).

Wenn Daten in den Ringbuffer geschrieben werden sollen, dann werden diese an die Position des Write-Pointers geschrieben. Danach wird der Write-Pointer so verschoben, dass er eine Stelle nach den geschriebenen Daten steht. Zwischen dem Read-Pointer und dem Write-Pointer stehen nun also die Nutzdaten. Es dürfen also nur diese Daten auch wieder ausgelesen werden, siehe Abbildung 2.

Wenn Daten ausgelesen werden, so startet der Lesevorgang beim Read-Pointer und darf maximal bis zum Write-Pointer erfolgen. Nach dem Lesevorgang wird der Read-Pointer auf eine Stelle nach dem zuletzt gelesenen Byte gesetzt, siehe Abbildung 3.

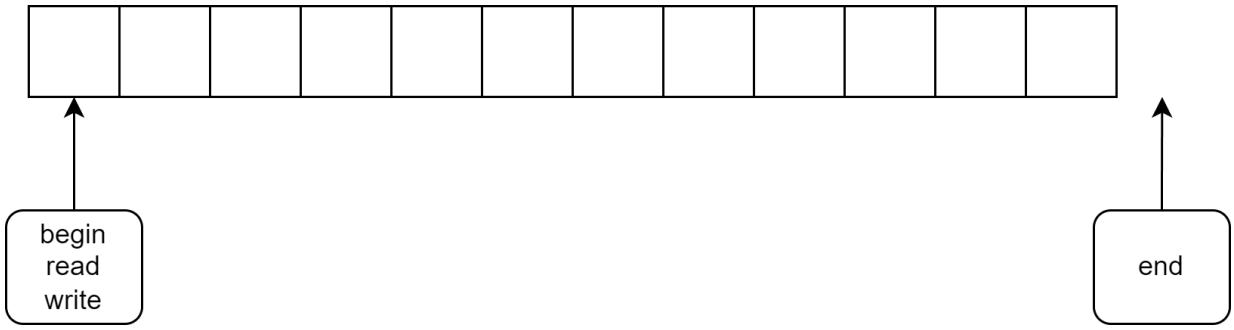

Abbildung 1: Ausgangssituation eines Buffers der Länge 12 Byte

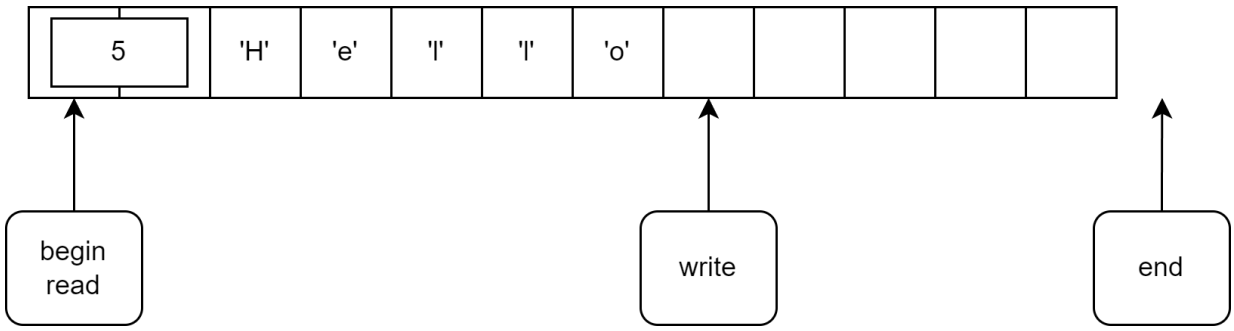

Abbildung 2: Ringbuffer nach Aufruf von `ringbuffer_write` mit der `message` „Hello“und einer `message_len` von 5. **Hier wird angenommen, dass 2 Byte für das Speichern der Länge genutzt werden. Das wird im echten Programm mehr sein!**

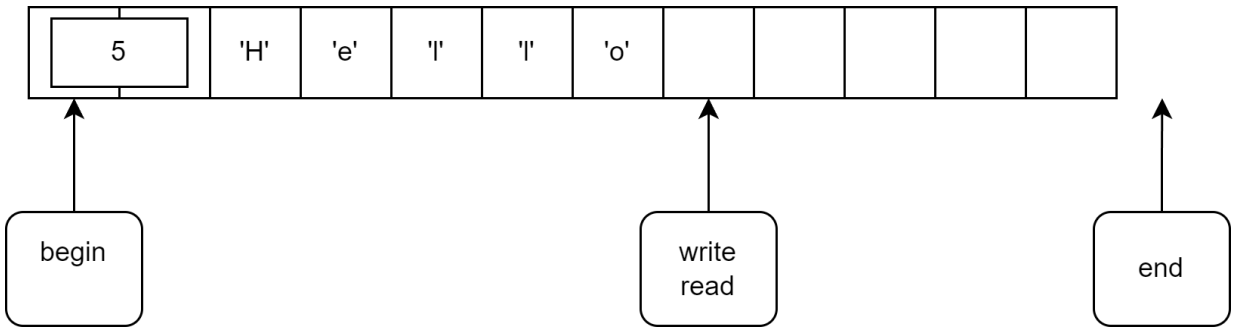

Abbildung 3: Ringbuffer nach Aufruf von `ringbuffer_read`. Der Parameter `buffer_len_ptr` muss mindestens 5 sein.

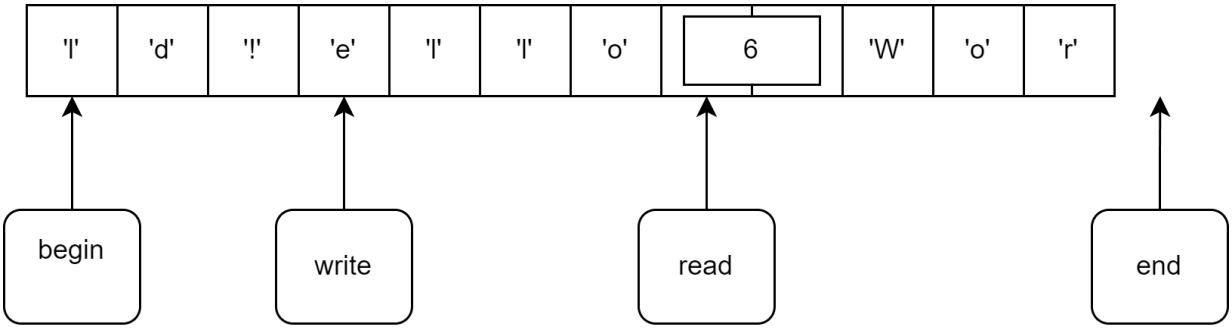

Abbildung 4: Ringbuffer nach Aufruf von `ringbuffer_write` mit der `message` „World!“und einer `message_len` von 6

Es kann vorkommen, dass der Platz zwischen Write-Pointer und Ende des Buffers kleiner ist, als die zu schreibende Nachricht. In diesem Fall soll der Schreibprozess bis zum Ende Buffers erfolgen und am Anfang des Buffers fortgesetzt werden (falls dies erlaubt d.h. es zu keiner Überschreibung kommen würde), siehe Abbildung 4. Ebenso muss auch der Leseprozess am Anfang fortgesetzt werden, wenn Nachrichten gelesen werden sollen, die wie oben beschrieben in den Buffer geschrieben wurden.

Auf diese Weise wandern die Read- und Write-Pointer immer weiter im Ring. Es darf allerdings niemals passieren, dass der Write-Pointer den Read-Pointer „überrundet“, da sonst Daten überschreiben werden, die zuvor noch nicht gelesen wurden. Ebenso darf auch der Read-Pointer den Write-Pointer nicht „überholen“, da sonst Daten gelesen werden, die bereits gelesen wurden.

Datenformat

Die Nachrichten sollen im folgenden Format im Ringbuffer gespeichert werden: Die ersten *Größe von size_t* Bytes dienen als Header bzw. Präfix der eigentlichen Nachricht. In diesen ersten Bytes speichern Sie die Länge der Nachricht. Hinter diesem Präfix speichern Sie die eigentliche Nachricht. Dieser Aufbau soll es später ermöglichen, Nachrichten variabler Länge beim Lesen zu erkennen und zu verarbeiten.

Zu implementierende Funktionen

Bitte entnehmen Sie Hinweise zu den Funktionen auch aus der Datei *lib/ringbuf.h*. Es folgen noch weitere Hinweise zu den Funktionen:

- `ringbuffer_init` Diese Funktion soll die Ausgangssituation wie in Abbildung 1 herstellen. Dazu wird ihr ein Pointer auf eine Ringbufferkontextstruktur übergeben, in der die relevanten Daten gespeichert werden sollen. Außerdem wird der Funktion die Speicherposition des Ringbuffers übergeben. Der Speicher für diesen Buffer muss schon zuvor reserviert sein.
- `ringbuffer_write` Diese Funktion führt die Write-Schritte, wie oben beschrieben aus.

- `ringbuffer_read` Dieser Funktion wird ein Buffer übergeben, in den die zu lesende Nachricht geschrieben werden soll. Es soll immer nur genau eine Nachricht gelesen werden. Dabei dürfen keine Nachrichten im Ringbuffer übersprungen werden. Der Parameter `buffer_len_ptr` gibt an, wie lang der übergebene Buffer ist und damit auch, wie lang die zu lesende Nachricht maximal sein darf. An die Speicheradresse, auf die diese Variable zeigt, soll am Ende des Lesevorgangs geschrieben werden, wie lang die gelesene Nachricht tatsächlich war.
- `ringbuffer_destroy` Zerstört Datenstrukturen, die während der Initialisierung angelegt wurden.

Bevor Sie mit der Bearbeitung beginnen, empfiehlt es sich, die möglichen Fälle zu überlegen, die auftreten können, wenn Nachrichten in den Ringbuffer geschrieben und aus ihm gelesen werden. Konkret sind dies die Fälle:

- **read_pointer** $>$ **write_pointer**: Der Lesepointer befindet sich hinter dem Schreibpointer.
- **read_pointer** $==$ **write_pointer**: Der Lesepointer ist gleich dem Schreibpointer.
- **read_pointer** $<$ **write_pointer**: Der Lesepointer befindet sich vor dem Schreibpointer.

Berücksichtigen Sie diese Fälle in Ihrer Implementierung. Um das Testen zu erleichtern, haben wir Ihnen bereits Testprogramme (in der Ordnern *test_unthreaded_no_wrap* und *test_unthreaded_wrap*) bereitgestellt.

Aufgabe 2: Threadsafer Ringbuffer (12P)

Erweitern Sie Ihren Ringbuffer in dieser Aufgabe so, dass mehrere Prozesse gleichzeitig auf den Ringbuffer zugreifen können. Überlegen Sie, welche kritischen Abschnitte in Ihrem Ringbuffer existieren und wie Sie diese schützen können.

Zur Umsetzung empfiehlt es sich zunächst, die Funktionen mit *lock* und *unlock* zu schützen. Später sollten Sie Ihre Lösung um *signal* und *wait* erweitern, um Busy-Waiting zu vermeiden.

Damit eine Write- oder Read-Operation nicht endlos blockiert, machen Sie sich mit der Funktion *pthread_cond_timedwait* vertraut. Diese Funktion ermöglicht es einem Thread, nach einer bestimmten Zeit aufzuwachen, auch wenn kein Signal gesendet wurde. Sollten Sie aufgrund eines Timeouts (wählen Sie hierfür eine Sekunde) aufwachen, können Sie die Funktion unter Rückgabe des entsprechenden Rückgabewerts verlassen.

Um das Testen zu erleichtern, haben wir bereits ein Testprogramm (im Ordner *test_threaded*) bereitgestellt.

Aufgabe 3: Simpler Netzwerkdaemon (6P)

Insbesondere im Bereich der Datenstromverarbeitung ist es häufig notwendig, effiziente Buffer (wie den von Ihnen implementierten thread-safen Ringbuffer) zu verwenden. In dieser Aufgabe sollen Sie einen simplen Netzwerkdaemon implementieren. Im grundlegenden Aufbau verarbeitet ein Netzwerkdaemon eingehende Nachrichten von verschiedenen Quellen, wie bspw. Netzwerkpakete von Port 80 (HTTP) oder 443 (HTTPS). Dazu speichert dieser die Pakete in einem Buffer, die dann von einem bzw. mehreren (Worker) Threads aus dem Ringbuffer gelesen und weiterverarbeitet werden. Im klassischen Fall handelt es

sich bei der Verarbeitung um Portweiterleitungen, Logging, Filterung, Firewalling, etc. In dieser Aufgabe geht es jedoch weniger um den Netzwerkaspekt, sondern vielmehr um das Zusammenspiel von Producern und Consumern (wie Sie es in der Vorlesung kennengelernt haben).

Verwenden Sie die vorgegebenen Datenstrukturen und Funktionen aus der Datei *daemon.c*.

Wir haben Ihnen bereits die in den Buffer schreibende Funktionalität bereitgestellt. Ihre Aufgabe besteht nun darin, die entsprechenden Processing-Threads (Reader) zu implementieren. Konkret bedeutet das, dass Sie sich um die korrekte Weiterleitung und ein einfaches Firewalling kümmern sollen.

Datenformat der Packets

Die Nachrichten die im Buffer gespeichert wurden, sind wie folgt aufgebaut: In den ersten *Größe von size_t* Bytes der Nachricht steht, von welchem Port die Nachricht stammt. In den darauffolgenden *Größe von size_t* Bytes steht, an welchen Port die Nachricht weitergeleitet werden soll. In den darauffolgenden *Größe von size_t* Bytes steht, um welches Packet der Ursprungsdatei es sich handelt. Dies wird dazu benötigt, um die Nachricht auf der Consumer-Seite wieder korrekt zusammenzusetzen. Die eigentliche Nachricht folgt nach diesen Metadaten.

Firewalling/Filtering

Sie sollen nun jene Nachrichten filtern, bei denen

- der Ursprungsport gleich dem Zielport ist.
- der Ursprungs- oder Zielport 42 sind.
- die Summe der Ursprungs- und Zielport 42 ist.
- Die eigentliche Nachricht den String *malicious* enthält. Dabei dürfen zwischen den Buchstaben des Strings beliebig viele andere Zeichen stehen. Die Reihenfolge der Buchstaben muss jedoch eingehalten werden. Die Groß- und Kleinschreibung soll dabei beachtet werden (d.h. *malicious* aber nicht etwa *Malicious* wird erkannt).

Weiterleitung

In dieser Aufgabe ist es nicht erforderlich, sich mit Netzwerkkommunikation (Ports, Sockets, usw.) auseinanderzusetzen. Stattdessen simulieren wir den Netzwerkverkehr über Dateien. Die schreibenden Threads schreiben Daten aus Dateien in den Ringbuffer, wobei Sie die Dateien in kleinere Pakete mit zufälligen Zeitintervallen versenden. Dadurch wird eingehender Netzwerkverkehr auf verschiedenen Ports simuliert. Die verarbeiteten Nachrichten müssen von Ihnen nicht an die tatsächlichen Zielports weitergeleitet werden. Stattdessen sollen Sie die Nachrichten an eine Ausgabedatei mit dem Namen *zielportnummer.txt* anhängen. Eine Nachricht die an den Port 2 gerichtet ist, soll an die Datei *2.txt* (ohne newline) angehängt werden. Dabei hängen sie nur den tatsächlichen Inhalt (nicht die Metadaten) an die Datei. Nachrichten, die gefiltert werden, also nicht weitergeleitet werden sollen, werden einfach verworfen.

Zur Vereinfachung des Testens haben wir bereits ein Testprogramm (im Ordner *test_daemon*) bereitgestellt.

Wichtige Überlegungen

- Denken Sie daran, die Thread-Sicherheit beim Schreiben in die Ausgabedateien zu gewährleisten. Zwei Threads die gleichzeitig in die gleiche Datei schreiben, können zu unvorhersehbaren Ergebnissen führen.
- Überlegen Sie, wie Sie damit umgehen, wenn mehrere Processing Threads, Pakete vom selbem Ursprungsport verarbeiten. In diesem Fall müssen die Pakete in der richtigen Reihenfolge in die Ausgabedatei geschrieben werden. Nutzen Sie hierfür das *lock*, *while*, *wait*, *work*, *unlock* Pattern, und machen Sie sich mit der Funktion *pthread_cond_broadcast* vertraut.
- Die Race Condition, die auftritt, wenn zwei verschiedene Quellen gleichzeitig an ein Ziel schreiben, kann ignoriert werden. Konkret am Beispiel: Wenn Port 1 zwei Packets (*ab*, *cd*) und Port 2 zwei Packets (*ef*, *gh*) an Port 3 senden, ist die Reihenfolge der geschriebenen Packets egal. Mögliche gültige Ausgaben sind: *abcdefgh*, *abefcdgh*, *abefghcd*, *efghabcd*, *efabghcd* und *efabcdgh*. Ungültig dagegen wäre z.B. *aefbcdgh*, da das Packet *ab* unterbrochen ist, und ebenso *cdabefgh*, da die Reihenfolge von *ab* und *cd* nicht korrekt ist. Wichtig ist, dass die Reihenfolge der Packets (innerhalb einer Quelle) korrekt bleibt und die Segmente nicht unterbrochen werden. Die Reihenfolge zwischen den Packets verschiedener Quellen kann jedoch beliebig sein.
- **Achtung:** Der Daemon ist bereits so implementiert, dass er nach Ablauf einer bestimmten Zeit (in diesem Fall 5 Sekunden) automatisch beendet wird. Dabei wird versucht, alle Threads ordnungsgemäß zu beenden, indem alle Threads mit *pthread_join* zusammengeführt werden. Da ihre Processing Threads jedoch in einer Endlosschleife laufen (sollen), wird diese Zeile nie erreicht. Um Ihre Threads von *außen* (also vom Hauptthread) zu beenden, nutzen wir die Funktion *pthread_cancel*. Machen Sie sich mit dieser Funktion und alle damit verbunden wie etwa *pthread_setcancelstate*, *pthread_setcanceltype* und *pthread_testcancel* vertraut. In Ihrer Abgabe müssen die Processing Threads mit *pthread_cancel* beendbar sein. Während der Entwicklung können Sie das Testprogramm (und die damit verbunden Processing Threads) auch mit *CTRL+C* manuell beenden, da es durchaus möglich ist, dass bereits Alles korrekt funktioniert, nur die Processing Threads noch nicht *von außen* beendbar sind.

Hinweise:

- **Beschreibungen der Funktionen:** Nähere Beschreibungen der einzelnen Funktionen finden Sie auch in den .h-Dateien im Ordner include.
- **Vorgaben:** Bitte ändern Sie bestehende Datenstrukturen, Funktionsnamen, Headerfiles, ... nicht. Eine Missachtung kann zu Punktabzug führen. Gerne können Sie weitere Hilfsfunktionen oder Datenstrukturen definieren, die Ihnen die Implementierung leichter gestalten.
- **Makefile:** Bitte verwenden Sie für diese Aufgabe das Makefile aus der Vorgabe. Führen Sie hierzu im Hauptverzeichnis `make` aus. *make* kompiliert das Projekt mit *clang* unter dem Ordner *build*. Die kompilierten Dateien können von einem Terminal aus in der

Shell ausgeführt werden. Um ein Programm auszuführen, verwenden Sie den folgenden Befehl: *./pathto/executable*. Beispiel für den simplen unthreaded Test: `./build/test_unthreaded_wrap/test_simple`.
Für Tests, bei denen Dateien als Argumente übergeben werden müssen, verwenden Sie: *./pathto/executable pathto/file1 pathto/file2*. Unter Ubuntu können *make* und *clang* mit dem Befehl `sudo apt install make clang` installiert werden. Durch den Befehl `make clean` werden kompilierte Dateien gelöscht. Gerne können Sie das Makefile um weitere Flags erweitern oder anderweitig anpassen. Ihr Programm sollte aber mit dem vorgegebenen Makefile weiterhin compilierbar bleiben. Auch hier sei erneut auf die README hingewiesen.

- **Memoryleaks:** Überprüfen Sie abschließend Ihr Programm auf Memory leaks, um zu evaluieren ob der gesamte vom Scheduler allokierte Speicher wieder freigegeben wurde. Hier empfiehlt sich das Kommandozeilenwerkzeug valgrind[1] bzw. leaks[2] unter MacOS. Memory leaks führen zu **Punktabzug**! Bedenken Sie, dass Sie valgrind auf die Ausführung von *./pathto/executable* anwenden müssen, **nicht** auf *make* (Sie wollen ja die Memory Leaks in Ihrem Programm finden, nicht in *make*.

Abgabe:

- **Verpacken** Sie den Ordner *src* mit den bearbeiteten **.c* Dateien (wie in der Vorgabe) zu einem zip-Archiv mit dem Namen *submission.zip*. Header-files (**.h*) sollen nicht verändert und damit auch nicht mitabgegeben werden. Hierfür können Sie das make target *make pack* verwenden. Damit *make pack* funktioniert muss das tool *zip* auf Ihrem System installiert sein! Das Archiv laden Sie dann in ISIS hoch.
- **Wichtig:** Es ist nicht notwenig das Archiv und dessen Inhalt mit Ihrem Namen bzw. Matrikelnummer zu personaliSieren. Die Abgabe wird automatisch Ihrem ISIS Account zugeordnet!

[1] http://valgrind.org/
[2] https://developer.apple.com/library/archive/documentation/Performance/Conceptual/ManagingMemory/Articles/FindingLeaks.html