Informatik 11
Embedded Software

RWTH AACHEN UNIVERSITY

Praktikum Systemprogrammierung

Versuch 5

Testtaskbeschreibung

Lehrstuhl Informatik 11 - RWTH Aachen

6. November 2023

Commit: 2f3d1172

Inhaltsverzeichnis

Dieses Dokument ist Teil der begleitenden Unterlagen zum *Praktikum Systemprogrammierung*. Alle zu diesem Praktikum benötigten Unterlagen stehen im Moodle-Lernraum unter https://moodle.rwth-aachen.de zum Download bereit. Folgende E-Mail-Adresse ist für Kritik, Anregungen oder Verbesserungsvorschläge verfügbar:

support.psp@embedded.rwth-aachen.de

5 Testtaskbeschreibung

5.1 Shared Access

Der Testtask *Shared Access* überprüft die Korrektheit der Implementierung der Schreib- und Lesefunktionen für den gemeinsamen Speicher. Dafür durchläuft der Testtask die folgenden zehn Phasen:

Phase 1 Es werden insgesamt fünf gemeinsame Speicherbereiche von je 10 Byte alloziert. Zusätzlich wird überprüft, ob diese korrekt angelegt werden.

Phase 2 Die zweite Phase unterteilt sich in 3 weitere Phasen 2a, 2b und 2c. Während der Phase 2a werden die zuvor allozierten fünf Speicherbereiche zum Schreiben und Lesen geöffnet und danach jeweils geschlossen. Zu beachten ist, dass die übergebene Adresse nicht unbedingt auf den Speicherbereich zeigt, der geöffnet bzw. geschlossen werden soll. Anschließend wird in Phase 2b und 2c bewusst ein Fehler hervorgerufen, indem mit Hilfe der Schreib- und Lesefunktionen für den gemeinsamen Speicher ein privater Speicherbereich ausgelesen respektive beschrieben werden soll.

Phase 3 Mit Hilfe der Funktion `os_sh_write` werden die fünf angelegten gemeinsamen Speicherbereiche komplett mit bestimmten Mustern beschrieben.

Phase 4 Mit Hilfe der Funktion `os_sh_read` werden die fünf gemeinsamen Speicherbereiche ausgelesen. Dabei wird überprüft, ob die aus der dritten Phase geschriebenen Werte noch mit den Originalwerten übereinstimmen. Ist dies nicht der Fall, wird eine entsprechende Fehlermeldung auf dem LCD ausgegeben.

Phase 5 Es wird bewusst ein Fehler verursacht, indem durch modifizierte Übergabeparameter der Funktion `os_sh_read` über die Grenze des zweiten gemeinsamen Speicherbereiches hinaus gelesen wird.

Phase 6 Hier wird ebenfalls bewusst ein Fehler hervorgerufen, indem durch modifizierte Übergabeparameter der Funktion `os_sh_write` über die Grenze des zweiten gemeinsamen Speicherbereiches hinaus geschrieben wird.

Phase 7 Es wird überprüft, ob der Anfang des dritten gemeinsamen Speicherbereiches nach den provozierten Fehlern nicht überschrieben wurde, also noch mit den Originalwerten aus Phase 3 übereinstimmt.

Phase 8 Das gleichzeitige Lesen zweier Prozesse wird simuliert, indem die Funktion `os_sh_readOpen` zweimal hintereinander aufgerufen wird. Außerdem wird geprüft, ob die Funktion `os_sh_readOpen` yieldet, wenn die maximale Anzahl an unterstützten gleichzeitigen Lesezugriffen oder ein Speicherbereich zum Beschreiben bereits geöffnet wurde. Des Weiteren wird überprüft, ob die Funktion `os_sh_writeOpen` yieldet, wenn bereits mindestens ein Speicherbereich zum Lesen geöffnet wurde.

Phase 9 In dieser Phase wird explizit geprüft, ob die Lese- und Schreibblockaden bei der Verwaltung von gemeinsamen Speicherbereichen eingehalten werden. Hierzu werden die Fälle *Read before write*, *Write before read* sowie *Write before write* nacheinander simuliert. Analog zu der achten Phase werden hierzu die Funktionen `os_sh_readOpen` und `os_sh_writeOpen` verwendet.

Phase 10 Hierbei wird geprüft, ob der Offset, welcher an die Funktionen `os_sh_read` und `os_sh_write` übergeben wird, korrekt umgesetzt wird. In mehreren Schritten wird ein Muster (1 bis 10) in einen gemeinsamen Speicherbereich geschrieben und im Anschluss daran auf verschiedene Arten wieder ausgelesen:

1. Zuerst wird die übergebene Speicheradresse im Speicherbereich verschoben und ein Byte ausgelesen. Hierbei sollte in allen Fällen der Wert des ersten Bytes (1) gelesen werden.

2. Im zweiten Schritt wird das Muster byteweise gelesen, indem der Offset in jedem Schritt um eins erhöht wird.

3. Der dritte Schritt liest den Speicherbereich vollständig aus und prüft das Ergebnisarray auf das korrekte Muster (1 bis 10).

4. Schließlich wird im vierten Schritt die Speicheradresse in jedem Schritt um eins erhöht und zusätzlich der übergebene Offset. Der Speicherbereich wird von dieser Position aus bis zum Ende ausgelesen und überprüft.

Es ist zu beachten, dass in der fünften und sechsten Phase jeweils ein Fehler provoziert wird. Demzufolge müssen entsprechende Fehlerausgaben auftreten und quittiert werden, woraufhin das Programm korrekt weiterlaufen soll.
Nachdem alle Phasen erfolgreich ausgeführt wurden, zeigt das LCD „`PRESS ENTER`" an. Nachdem dies mit einem Tastendruck bestätigt wurde, wird „`TEST PASSED`" und „`WAIT FOR IDLE`" angezeigt und der Testtask terminiert. Im Anschluss muss der Leerlaufprozess ausgeführt werden.

Fehlermeldungen

„`1. Allocating`"
Zwei oder mehr der in Phase 1 allozierten gemeinsamen Speicherbereiche sind der gleiche Speicherbereich.

„x_1 FAILURE @ x_2/10"
Das gelesene Byte x_2 in Chunk x_1 entspricht nicht dem zuvor dort geschriebenen Byte.

„7. Checking"
Der Speicher wurde trotz access violation verändert.

„Not enough memory"
Es konnte nicht genug Speicher alloziert werden.

„Address has changed"
Die Adresse eines gemeinsamen Speicherbereichs ist nach dem Öffnen anders.

„No yield when (read|w/r|r/w) opened"
Die Funktion `os_sh_readOpen` bzw. `os_sh_writeOpen` verwendet `os_yield` nicht (richtig). Beispielsweise gibt `os_sh_readOpen` stattdessen 0 zurück, falls ein Speicherbereich bereits so oft geöffnet ist, wie maximal unterstützt wird.

„FAILURE @ x"
Die Barrieren zur Vermeidung von Zugriffskonflikten funktionieren nicht richtig. x gibt dabei an welcher Test in Phase 9 fehlgeschlagen ist. Beispielsweise „`Read before write`".

„Pattern mismatch"
Die mit `os_sh_read()` in Phase 10 gelesenen Daten entsprechen nicht den zuvor in den Speicherbereich geschriebenen Daten.

„Unexpected error"
Ein Fehler ist aufgetreten, obwohl kein Fehler auftreten sollte.

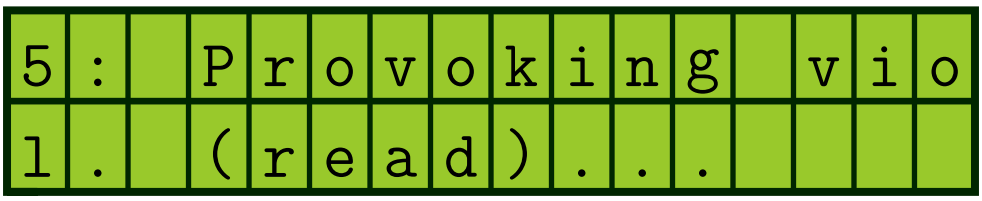

Abbildung 5.1: Shared Access. Ausgabe während des Testes.

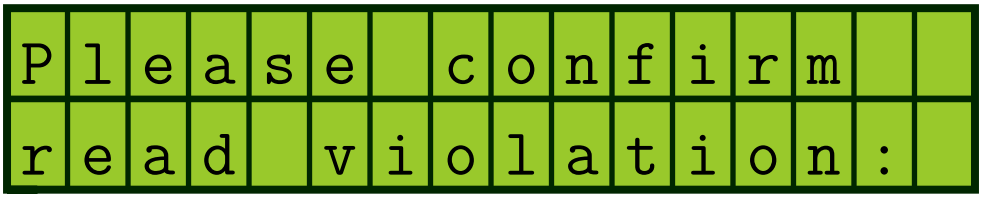

Abbildung 5.2: Shared Access. Ausgabe während des Testes.

5.2 Free Private

Der Testtask *Free Private* überprüft, ob die Restriktionen bezüglich der Freigabe von privaten und gemeinsamen Speicherbereichen eingehalten wurden. Dazu alloziert der Testtask einen privaten und einen gemeinsamen Speicherbereich. Darauf wird versucht den privaten Speicherbereich mittels `os_sh_free` freizugeben, was durch die LCD-Ausgabe aus Abbildung 5.3 signalisiert wird. Danach wird versucht den gemeinsamen Speicherbereich mittels `os_free` freizugeben, was wiederum durch die LCD-Ausgabe aus Abbildung 5.4 signalisiert wird. Ein solcher Speicherfreigabeversuch muss jeweils als Fehler erkannt und auf dem LCD angezeigt werden.
Nachdem alle Phasen erfolgreich ausgeführt wurden, zeigt das LCD „`PRESS ENTER`" an. Nachdem dies mit einem Tastendruck bestätigt wurde, wird „`TEST PASSED`" und „`WAIT FOR IDLE`" angezeigt und der Testtask terminiert. Im Anschluss muss der Leerlaufprozess ausgeführt werden.

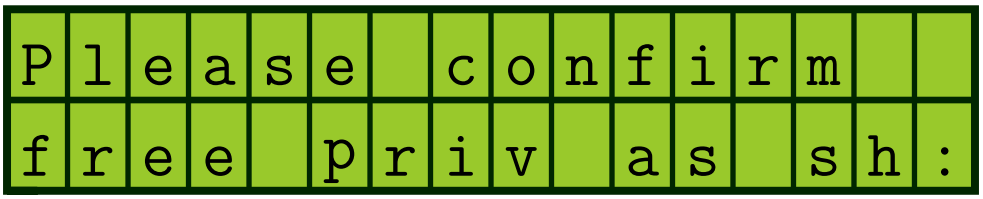

Abbildung 5.3: `os_sh_free` angewandt auf einen privaten Speicherbereich

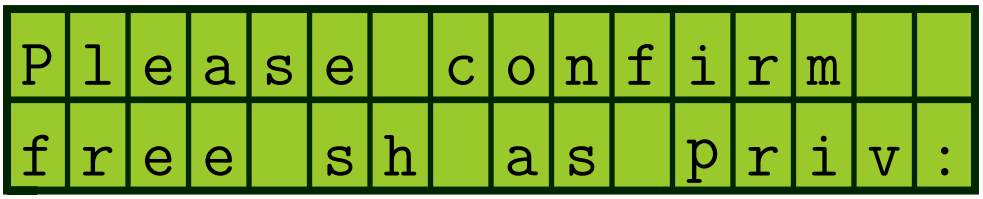

Abbildung 5.4: `os_free` angewandt auf einen gemeinsamen Speicherbereich

5.3 Yield

Der Testtask *Yield* ist in fünf Phasen unterteilt und überprüft, ob die Funktion `os_yield` korrekt implementiert wurde.

Phase 1 In dieser Phase wird das Interrupt-Enable-Flag im `SREG` vom Testprozess explizit ausgeschaltet, welcher daraufhin einen weiteren Prozess startet und seine Rechenzeit mittels `os_yield` abgibt. Der soeben gestartete Prozess setzt das Global-Interrupt-Enable-Bit und terminiert. Anschließend wird überprüft, ob die Funktion `os_yield` das `SREG` korrekt auf den zuvor abgeschalteten und gespeicherten Wert zurücksetzt.

Phase 2 Es wird die Geschwindigkeit des implementierten `os_yield` inklusive Schedulers getestet und anschließend ausgegeben. Die Geschwindigkeit wird nachher in Phase 3 genutzt, um künstlichen Rechenaufwand von äquivalenter Länge zu erzeugen. In dieser Phase führt jede noch so langsame Implementierung zum bestehen, es sollte aber beachtet werden, das der gesamte Yield-Test 5 Minuten nicht übersteigen darf. Sollte Ihre gemessene Yield-Dauer um mehrere Größenordnungen von 3150us abweichen, so kann dies zu einem nicht bestehen des Tests führen.

Phase 3 Es werden die Iterationen der Programmdurchläufe von zwei verschiedenen Prozessen gezählt, wovon nur ein Prozess die Funktion `os_yield` nutzt. Diese Iterationen und deren Verhältnis werden in der unteren Zeile des LCDs für jede Strategie angezeigt. Das Verhältnis sollte sich innerhalb einer kurzen Zeitspanne einpendeln und dann konstant bleiben (vgl. Abbildung 5.5).
Da beide Prozesse durch die Laufzeitmessung aus Phase 2 annähernd gleiche Laufzeiten haben, sollten alle Strategien gegen das Verhältnis von 2 konvergieren. Um die Laufzeit des Tests nicht unnötig zu erhöhen werden die einzelnen Strategietests nicht bei erreichen des Wertes 2 sondern nach einer festen Anzahl Iterationen beendet. Es werden hierbei alle definierten Strategie auf ihr Verhältnis hin untersucht.
Da bei der Strategie `OS_SS_RUN_TO_COMPLETION` kein Verhältnis errechnet werden kann, wird sie in dieser Phase übersprungen.

Phase 4 Hier wird für jede Scheduling-Strategie der Sonderfall getestet, das nur ein einziger Prozess neben dem Leerlaufprozess existiert und dieser `os_yield` aufruft. Alle Strategien sollten den aktuellen Prozess erneut auswählen um die Rechenzeit effizient zu nutzen.

Phase 5 Die letzte Phase deckt einen Sonderfall der Strategie `OS_SS_RUN_TO_COMPLETION` ab. Rechenzeit soll bis zur Terminierung oder eigenen Aufgabe mittels `os_yield` abgetreten werden. Dies bedeutet das eine Rückkehr zum Testprogramm erst erfolgen darf nachdem der nebenläufig gestartete Hilfsprozess terminiert ist.

Nachdem alle Phasen erfolgreich ausgeführt wurden, zeigt das LCD „`PRESS ENTER`" an. Nachdem dies mit einem Tastendruck bestätigt wurde, wird „`TEST PASSED`" und „`WAIT FOR IDLE`" angezeigt und der Testtask terminiert. Im Anschluss muss der Leerlaufprozess ausgeführt werden.

	Verhältnisse non-Yield/Yield					
	Even	MLFQ	Random	Inactive aging	Round Robin	Run to completion
MIN	2	2	2	2	2	X
MAX	255	255	255	255	255	X

Tabelle 5.1: Grenzwerte für das Bestehen des Testtasks.

Hinweis

- Die Verhältnisse sind sehr anfällig gegenüber den Optimierungen des Compiler. Beim absolvieren des Testtask sollten diese daher aktiviert sein (Begleitendes Dokument, 3.4).
- Es wurde bewusst darauf verzichtet am Ende von Phase 1 das `GIEB` erneut zu aktivieren. Alle while-Schleifen in `os_kill` müssen bereits durch `os_yield` ersetzt worden sein. Anderenfalls wird der Testtask nicht mit Phase 2 fortfahren.
- Ein Verhältnis von `1/1` ist bei allen Strategien inkorrekt (vgl. Tabelle 5.1).
- Die Werte aus Tabelle 5.1 führen zum bestehen des Testes und sind inklusive.

Fehlermeldungen

„SREG"
Das Global Interrupt Enable Bit im SREG wurde nach einem Aufruf von `os_yield` und anschließender Rückkehr nicht korrekt wiederhergestellt.

„INVERSION"
Während der Testausführung wurde dem Prozess der `os_yield` aufruft mehr Rechenzeit zugeteilt als anderen Prozessen. Die grundlegende Funktion von `os_yield` ist nicht erreicht.

„Q-UNDER"
Eine Strategie (2. Zeile LCD) hat die zulässigen Grenzwerte in Phase 3 unterschritten.

„Q-OVER"
Eine Strategie (2. Zeile LCD) hat die zulässigen Grenzwerte in Phase 3 überschritten.

„TIMEOUT"
Der Test hat die maximal zulässige Laufzeit überschritten.

„IDLE HIT"
Eine Strategie (2. Zeile LCD) hat in Phase 4 anstatt dem aktuell einzigen Prozess im Zustand `OS_PS_BLOCKED` den Leerlaufprozess ausgewählt.

„RCOMP-TIME"
Bei Nutzung der Strategie `OS_SS_RUN_TO_COMPLETION` konnte der Testprozess seine Rechenzeit nicht dauerhaft an einen anderen Prozess bis zu dessen Terminierung mittels `os_yield` abtreten.

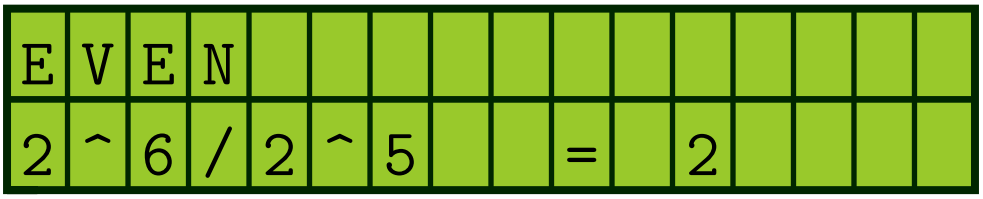

Abbildung 5.5: Yield. Ausgabe während des Testes.

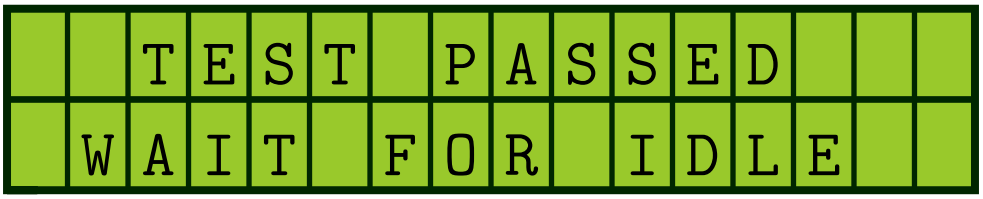

Abbildung 5.6: Yield. Ausgabe bei erfolgreichem Test.

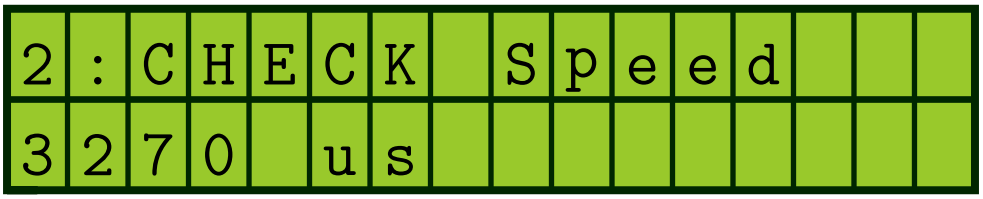

Abbildung 5.7: Yield. Ergebnis Phase 2

5.4 Stability Shared

Der Testtask *Stability Shared* überprüft, ob das Allozieren, Beschreiben und Lesen von gemeinsamen Speicherbereichen korrekt funktioniert.

Ein Prozess alloziert einen gemeinsamen Speicherbereich und überprüft durchgehend den Inhalt vom Anfang und Ende des Speicherbereiches miteinander. Zusätzlich werden drei Prozesse gestartet, die unabhängig voneinander aus dem gemeinsamen Speicher lesen, die gelesenen Werte überprüfen und ihn neu beschreiben. Kommt es hier zu der Fehlermeldung „`Write was interleaved`", so wurden die typischen Konflikte von gemeinsamen Speicher nicht vollständig beachtet. Des Weiteren findet eine Neupositionierung statt, welche den Inhalt und die Position des gemeinsamen Speicherbereiches verändert.
Auf dem LCD wird währenddessen in der oberen Zeile die verstrichene Zeit angezeigt und in der unteren Zeile links eine Zahl inkrementiert, welche die Anzahl der erfolgreichen wechselseitigen Ausschlüsse angibt. Erhöht sich diese Zahl nicht, so ist der wechselseitige Ausschluss nicht gegeben. Eine Ursache könnten nicht geschlossene oder überflüssige kritische Sektionen sein. In der unteren Zeile rechts werden die Ziffern von drei bis fünf durchgehend angezeigt. Diese entsprechen jeweils der ID des aktuellen Prozesses. Die

genaue Reihenfolge ist hier irrelevant, jedoch muss jede Ziffer von drei bis fünf immer wieder auftreten. Wenn die Prozess-IDs nicht variieren, liegt mit großer Wahrscheinlichkeit ein Deadlock aufgrund nicht geschlossener Chunks vor. Eine Ursache dafür kann eine fehlerhafte Dereferenzierung des Pointers auf die Mapadresse sein.
Dieser Testtask gilt als bestanden, wenn nach drei Minuten keine Fehler ausgegeben wurden, die Zahl unten links sich weiterhin erhöht und die Ziffern von drei bis fünf weiterhin ausgegeben werden.

Fehlermeldungen

„Write was interleaved"
Ein Chunk wurde gleichzeitig von mehreren Prozessen beschrieben.

5.5 Stability Shared External

Der Testtask *Stability Shared External* besitzt den gleichen Funktionsumfang wie *Stability Shared*, testet jedoch die Verwaltung von gemeinsamen Speicher auf dem externen SRAM. Demzufolge gilt dieser Testtask ebenfalls als bestanden, wenn nach drei Minuten keine Fehler ausgegeben wurden, die Zahl unten links sich weiterhin erhöht und die Ziffern von drei bis fünf weiterhin ausgegeben werden.

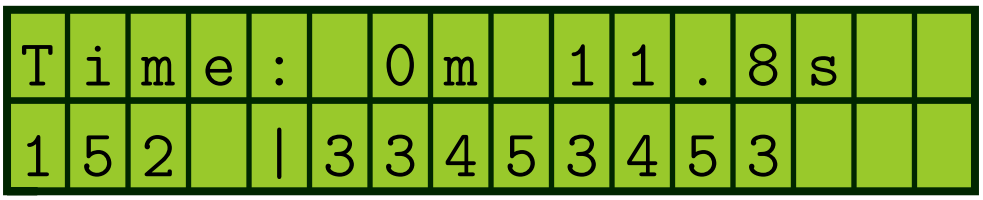

Abbildung 5.8: Stability Shared. Ausgabe während des Testes.

5.6 Process Queues

Mithilfe des Testtasks *Process Queues* kann die Funktionalität der Queues, die für MLFQ benötigt werden, isoliert getestet werden. Es bietet sich daher an, diesen Test vor dem *Multilevel-Feedback-Queue* Test durchzuführen und etwaige Fehler zu beseitigen, da dies das Debuggen von MLFQ erleichtert. Der Testtask durchläuft dabei fünf Phasen. Die ersten vier Phasen testen dabei nur die `pqueue_`-Funktionen isoliert. Die letzte Phase testet die Initialisierung der Queues im Kontext von MLFQ und erfordert daher eine korrekte Implementierung von `os_setSchedulingStrategy`.

Phase 1: Init In der ersten Phase wird die Methode `pqueue_init` auf einer `ProcessQueue` aufgerufen und dann einige grundlegende Parameter der Queue abgefragt.

Phase 2: Simple Operations In der zweiten Phase wird ein Element mittels `pqueue_append` eingefügt, dessen Wert dann mit `pqueue_getFirst` abgefragt und dieses dann schließlich mit `pqueue_dropFirst` entfernt. Vor und nach jeder Operation wird mit `pqueue_hasNext` getestet, ob Elemente in der Queue sind.

Phase 3: Consistency In der dritten Phase wird geprüft, ob die Queue die Prozess-IDs aller möglichen Prozesse ohne Datenverlust abspeichern kann. Dazu wird die Startposition von `queue.head` und `queue.tail` über das ganze Array variiert, um mögliche Fehler beim Wrap-Around abzufangen. Es wird in jedem Durchlauf die Folge $1, 2, \ldots$ `MAX_NUMBER_OF_PROCESSES` $- 1$ in die Queue eingespeichert. Dann werden alle Elemente nacheinander entfernt, wobei überprüft wird, ob die PID-Folge immer noch korrekt ist.

Phase 4: Remove Die vierte Phase prüft die Funktion `pqueue_removePID`. Zunächst wird jeweils mit einer vollen Queue geprüft, ob das erste, eines der mittleren und das letzte Element entfernt werden können. Anschließend werden aus einer vollen Queue sukzessive die Elemente entfernt, bis die Queue leer ist.

Phase 5: MLFQ Integration In dieser Phase wird zunächst sichergestellt, dass neben dem Idle-Prozess die folgenden Prozesse mit den entsprechenden Prioritäten existieren:

PID	1	2	3	4
Priorität	`0b00000000`	`0b01000000`	`0b10000000`	`0b11000000`

Anschließend wird mittels Aufruf von `os_setSchedulingStrategy` die Strategie auf MLFQ gesetzt, was dazu führen sollte, dass die oben genannten Prozesse entsprechend ihrer Prioritäten in die Queues sortiert werden. Es wird anschließend geprüft, dass sich in den vier Prozess-Queues jeweils ein Prozess befindet, wobei die Prozesse jeweils in auf- oder absteigender Reihenfolge in den vier Queues liegen dürfen. Zuletzt werden Prozesse 2-4 mit `os_kill` beendet und die Strategie wieder auf Even gewechselt. Diese Testphase wird als kritische Sektion ausgeführt, wodurch sichergestellt ist, dass die MLFQ-Strategie nie tatsächlich ausgeführt wird.

Um den Ablauf des Testtasks besser nachzuvollziehen wird der Zustand der getesteten Queue nach jeder Operation auf dem Bildschirm angezeigt: In der oberen Zeile des Bildschirms wird mit „`[`" und „`]`" begrenzt das Array `queue.data` teilweise dargestellt (nur Zahlen zwischen `queue.head` und `queue.tail` werden ausgegeben). Ein Wert, welcher keiner gültigen Prozess-ID entspricht und innerhalb des Queue-Bereichs liegt, deutet auf einen Fehler hin und wird daher mit „`X`" markiert. In der zweiten Zeile werden die Indizes `queue.head` mit einem „`^`" und `queue.tail` mit einem „`.`" dargestellt. Sollte der Test aufgrund eines Queue-bezogenen Fehlers fehlschlagen, kann man mit einer beliebigen Taste zwischen der Fehlermeldung und der Anzeige des Queue-Zustands hin- und herwechseln.

Nachdem alle Phasen erfolgreich ausgeführt wurden, zeigt das LCD „PRESS ENTER" an. Nachdem dies mit einem Tastendruck bestätigt wurde, wird „TEST PASSED" und „WAIT FOR IDLE" angezeigt und der Testtask terminiert. Im Anschluss muss der Leerlaufprozess ausgeführt werden.

Fehlermeldungen

„Head/Tail not 0"
Nach der Initialisierung ist `queue.head` oder `queue.tail` nicht auf 0 gesetzt.

„size too small for MLFQ"
Das `size`-Attribut gibt eine Größe an, die zu klein ist, die PIDs aller nicht-Idle-Prozesse zu speichern. Damit ist die Queue nicht zur Implementierung von MLFQ geeignet.

„array too small for size"
Das `queue.data` Array enthält weniger als `queue.size` Elemente und kann daher die gegebene Größe nicht realisieren.

„queue should have next"
`pqueue_hasNext` hat 0 zurückgegeben, obwohl Elemente in der Queue enthalten sein sollten.

„unexpected element"
Das mittels `pqueue_getFirst` abgefragte Element hat nicht den erwarteten Wert.

„getFirst changed queue"
Die Methode `pqueue_getFirst` hat den Zustand der Queue verändert.

„queue should be empty"
`pqueue_hasNext` hat 1 zurückgegeben, obwohl die Queue leer sein sollte.

„Remove error: first"
Fehler beim Entfernen des ersten Elements

„Remove error: mid"
Fehler beim Entfernen eines Elements aus der Mitte

„Remove error: last"
Fehler beim Entfernen des letzten Elementes

„Remove error: successive"
Fehler beim mehrmaligen Entfernen von Elementen

„Incorrect PID"
Einer der Prozesse für den Integrationstest hat eine unerwartete PID.

„Incorrect queue (press UP)"
Die Prozess-Queues sind nach dem Ändern der Strategie nicht im erwarteten Zustand. Mit den Tasten `UP` und `DOWN` kann zwischen der Fehlermeldung und den vier Queues hin- und hergeblättert werden.

„Kill Failure"

`os_kill` hat beim Beenden eines fremden Prozesses `false` zurückgeliefert.

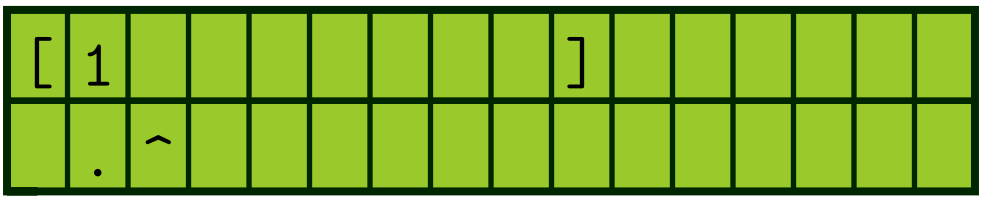

Abbildung 5.9: Ein Objekt in der Queue.

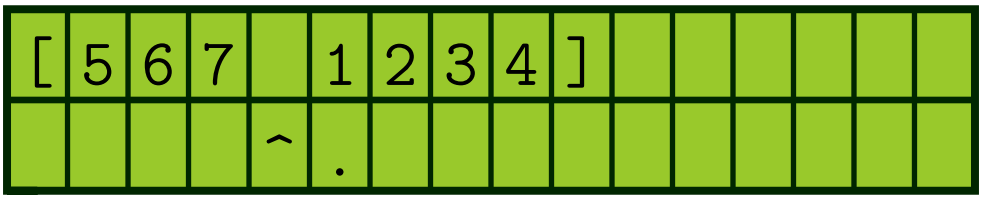

Abbildung 5.10: Eine Queue mit Wrap-Around

5.7 Multilevel-Feedback-Queue

Mithilfe des Testtasks *Multilevel-Feedback-Queue* lässt sich überprüfen, ob die Schedulingstrategie korrekt implementiert wurde. Hierzu werden die Programm-IDs der Prozesse in der Reihenfolge ausgegeben, in welcher sie vom Scheduler Rechenzeit zugewiesen bekommen. Dabei können Prozesse, im Gegensatz zu dem Testtask für die anderen Schedulingstrategien, sowohl Terminieren als auch Rechenzeit vorzeitig abgeben. In letzterem Fall wird anstatt der Programm-ID ein korrespondierender Buchstabe ausgegeben. So steht „`a"` für Programm 1, „`b"` für Programm 2, usw..

Erster Durchlauf:	„`12343b44233455g474d2222651111111"`
Zweiter Durchlauf:	„`144444444111111111111111111111"`

Bei einer falschen Implementierung bleibt die Ausgabe sichtbar und die erste falsche Stelle wird markiert. Nachdem alle Phasen erfolgreich ausgeführt wurden, zeigt das LCD „`PRESS ENTER"` an. Nachdem dies mit einem Tastendruck bestätigt wurde, wird „`TEST PASSED"` und „`WAIT FOR IDLE"` angezeigt und der Testtask terminiert. Im Anschluss muss der Leerlaufprozess ausgeführt werden.

Das Testszenario sieht dabei wie in der nachfolgenden Tabelle 5.2 aus. Zu Beginn wird Programm 1 mit der Default Priorität gestartet. Im ersten Rechenslot des Prozesses startet dieser nun erst Programm 2 und anschließend Programm 3 („`1: 2,3"`). Danach wird

`os_setSchedulingStrategy` für die Multilevel-Feedback-Queue aufgerufen. Alle Zeitangaben beziehen sich auf die Laufzeit des jeweiligen Prozesses. Das bedeutet, Programm 2 gibt in seinem zweiten Rechenslot vorzeitig Rechenzeit ab und nicht schon nach dem zweiten Scheduleraufruf. Eine weitere Besonderheit ergibt sich im zehnten Rechenslot von Programm 1, welches die Ausführung von Programm 4 durch Aufruf der Funktion os_kill vorzeitig beendet. Nach der Terminierung des zugehörigen Prozesses, wird die erfolgreiche Eingliederung der maximalen Anzahl von Instanzen des Programms 2 in die Warteschlangen überprüft. Dies geschieht durch entsprechende Aufrufe der Funktion `os_exec`. Danach werden die neu erzeugten Prozesse terminiert und die Ausführung von Programm 1 fortgesetzt.

Fehlermeldungen

„Program 4 not in slot 4"
Instanz des Programs mit ID 4 wurde nicht im erwarteten Slot des Arrays `os_processes` abgelegt.

„Could not exec process"
Das Erstellen von mehreren Instanzen des Programms 2 war nicht erfolgreich.

„Queue incorrect"
Nach der Eingliederung der erzeugten Prozesse wurden falsche Einträge in der Warteschlange vorgefunden.

„Could not kill process"
Das Terminieren der zusätzlich erzeugten Prozesse ist fehlgeschlagen.

Programm	Priorität	Laufzeit	Yield	Startet	os_kill
1	0b00000010	∞	-	1: 2,3	10 : 4
2	0b11000000	7	2		
3	0b10000000	4	-	1: 4	
4	0b11000000	∞	7	4: 5,6	
5	0b10000000	3	-	1: 7	
6	0b01000000	1	-	-	
7	0b10000000	2	1	-	

Tabelle 5.2: Beschreibung des Testszenarios