Informatik 11
Embedded Software

RWTH AACHEN UNIVERSITY

Praktikum Systemprogrammierung

Versuch 5

Testtaskbeschreibung

Lehrstuhl Informatik 11 - RWTH Aachen

6. Dezember 2019

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:

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

2. Phase 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.

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

4. Phase 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.

5. Phase 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.

6. Phase 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.

7. Phase 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.

8. Phase 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.

9. Phase 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.

10. Phase 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:
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.
Im zweiten Schritt wird das Muster byteweise gelesen, indem der Offset in jedem Schritt um eins erhöht wird.
Der dritte Schritt liest den Speicherbereich vollständig aus und prüft das Ergebnisarray auf das korrekte Muster (1 bis 10).
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. Tritt in den anderen Phasen keine Fehlermeldung auf, so wird am Ende des Testtasks `„All tests passed"` auf dem LCD ausgegeben. Im Anschluss daran terminiert der Testtask, sodass der Leerlaufprozess aktiv werden sollte.

Fehlermeldungen

FAILURE
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.

FAILURE @ Checking
Der Speicher wurde trotz access violation verändert.

Not enough memory
Es konnte nicht genug Speicher alloziert werden.

Address shouldn't have 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.

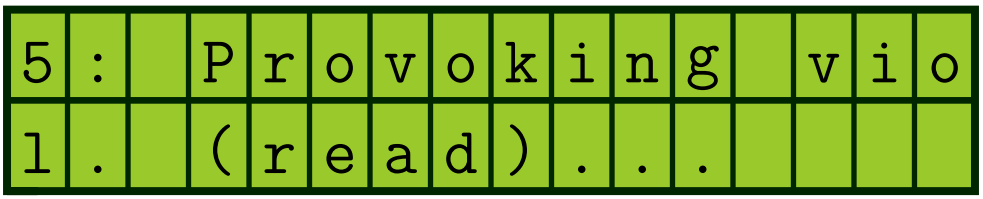

Abbildung 5.1: Shared Access. Ausgabe während des Testes.

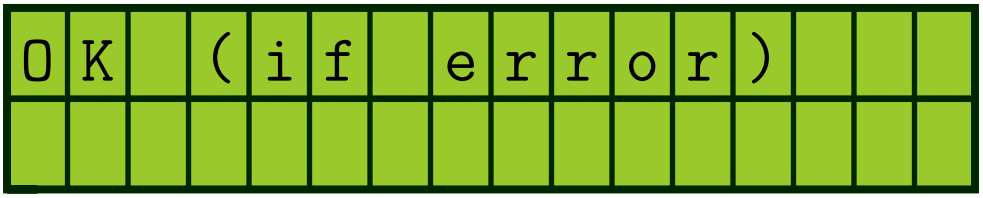

Abbildung 5.2: Shared Access. Ausgabe während des Testes.

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 **??** signalisiert wird. Danach wird versucht den gemeinsamen Speicherbe-

reich mittels `os_free` freizugeben, was wiederum durch die LCD-Ausgabe aus Abbildung **??** signalisiert wird. Ein solcher Speicherfreigabeversuch muss jeweils als Fehler erkannt und auf dem LCD angezeigt werden.

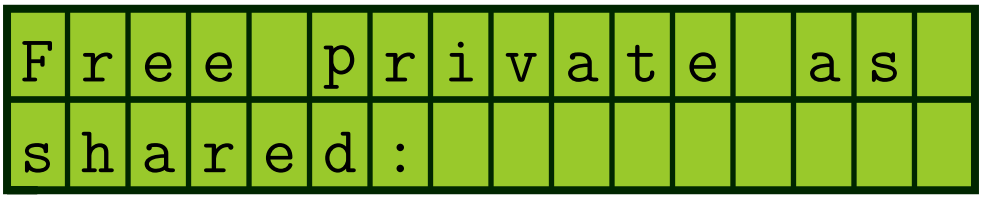

Abbildung 5.3: `os_sh_free` angewandt auf einen privaten Speicherbereich

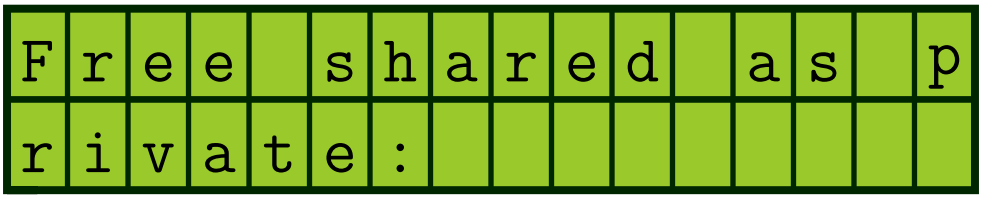

Abbildung 5.4: `os_free` angewandt auf einen gemeinsamen Speicherbereich

5.3 Yield

Der Testtask *Yield* ist in zwei Phasen unterteilt und überprüft, ob die Funktion `os_yield` korrekt implementiert wurde.
In der ersten Phase wird das Interrupt-Enable-Flag im `SREG` vom Testprozess explizit ausgeschaltet, welcher daraufhin einen weiteren Prozess startet und seine Rechenzeit abgibt. Der soeben gestartete Prozess setzt das Interrupt-Enable-Flag und terminiert. Anschließend wird überprüft, ob die Funktion `os_yield` das `SREG` korrekt zurücksetzt.
In der zweiten Phase 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 angezeigt. Dieses Verhältnis sollte sich innerhalb einer kurzen Zeitspanne einpendeln und dann konstant bleiben (vgl. Abbildung **??**). Es werden hierbei alle defnierten Strategie auf ihr Verhältnis hin untersucht. Da bei der Strategie `OS_SS_RUN_TO_COMPLETION` kein Verhältniss errechnet werden kann, wird nur ein einfaches `os_yield` getestet. Ist dies erfolgreich, wird 1 zurückgegeben, sonst 255, was zu einem Fehler führt.

Hinweis: Ein Verhältnis von `1/1` ist bei allen Strategien außer `OS_SS_RUN_TO_COMPLETION` inkorrekt (vgl. Tabelle **??**).

Hinweis: Besonderes Augenmerk sollte auch auf `OS_SS_RANDOM` gelegt werden, da ein erfolgreiches `os_yield` mit Verhältniss größer eins nicht genügt den Test zu bestehen sondern mindestens größer 2 sein muss um eine Pseudo-random-verteilung zu gewährleisten (vgl. Tabelle **??**).

Hinweis: Folgen Werte führen zum bestehen des Testes und sind exklusiv:

	Verhältnisse nonYield/Yield					
	Even	MLFQ	Random	Inactive ageing	Round Robin	Run to completion
MIN	4	30	3	4	8	1
MAX	255	255	255	255	255	1

Tabelle 5.1: Grenzwerte für das Bestehen des Testtasks.

Fehlermeldungen

SREG not restored
Das Global Interrupt Enable Bit im SREG wurde nach einem Aufruf von `os_yield` nicht korrekt wiederhergestellt.

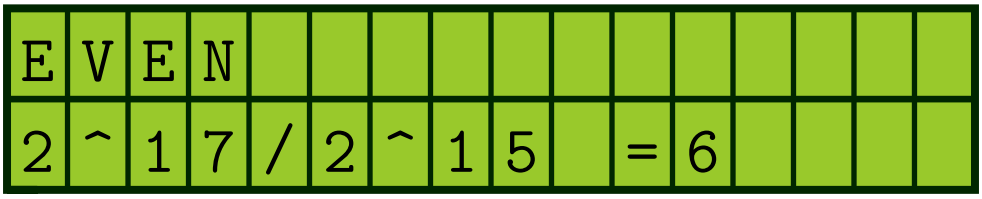

Abbildung 5.5: Yield. Ausgabe während des Testes.

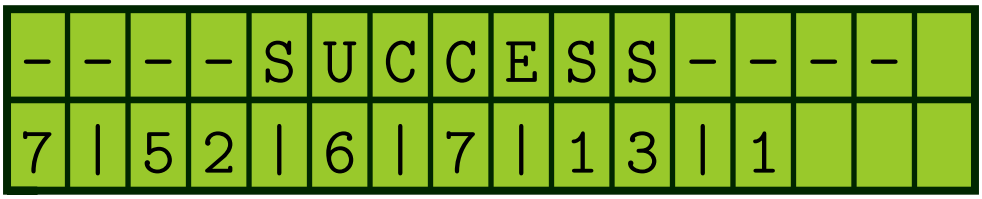

Abbildung 5.6: Yield. Ausgabe bei erfolgreichem Test.

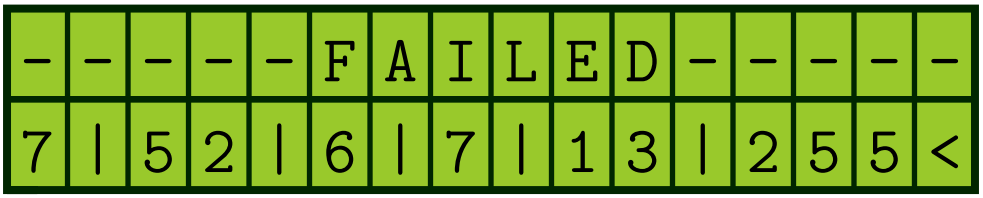

Abbildung 5.7: Yield. `OS_SS_RUN_TO_COMPLETION` konnte kein `os_yield` durchführen.

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 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 und die Ziffern von drei bis fünf weiterhin ausgegeben werden.

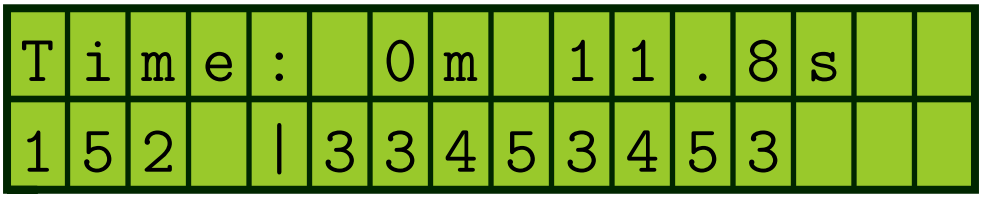

Abbildung 5.8: Stability Shared. Ausgabe während des Testes.

5.6 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..
Bei korrekter Implementierung ergibt sich beim Durchlauf des Testtasks folgende Ausgabe, jeweils gefolgt von einem „`OK`" und einem abschließenden „`Test passed.`":

Erster Durchlauf:	`12343b44233455g474d2222651111111`
Zweiter Durchlauf:	`1444444441111111111111111111111`

Bei einer falschen Implementierung bleibt die Ausgabe sichtbar und die erste falsche Stelle wird markiert.

Das Testszenario sieht dabei wie in der nachfolgenden Tabelle **??** 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

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.

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 startet 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.