Informatik 11
Embedded Software

RWTH AACHEN UNIVERSITY

Praktikum Systemprogrammierung

Versuch 2

Testtaskbeschreibung

Lehrstuhl Informatik 11 - RWTH Aachen

27. Oktober 2023

Commit: f727c3c0

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

2 Testtaskbeschreibung

Die Testtasks können Sie einbinden, indem Sie die jeweilige `progs.c`-Datei in das Projekt einbinden. Bitte beachten Sie, dass nicht mehrere Testtasks gleichzeitig geladen werden können, da sonst mehrere Programme mit der selben ID definiert werden. Beachten Sie außerdem, dass in Versuch 2 in keinem Testtask der Leerlaufprozess laufen sollte, da die Testtasks immer *Ready* sind. Sollte in einem der untenstehenden Testtasks der Leerlaufprozess ausgewählt werden gilt dieser als nicht bestanden.

2.1 Unittests

Die im Folgenden beschriebenen Tests sind Unittests. Dies bedeutet, dass sie getrennt vom restlichen Betriebssystem ausgeführt werden. Beachten Sie bitte, dass sie dafür noch vor der `main`-Funktion ausgeführt werden.

2.1.1 `os_exec`

Der Unittest `os_exec` überprüft zunächst, ob `os_exec()` als Rückgabewert `INVALID_PROCESS` liefert, wenn alle Slots des Prozess-Arrays belegt sind oder ein ungültiges Programm übergeben wird. Falls `os_exec` dies nicht abfängt, wird eine entsprechende Fehlermeldung, wie in Abbildung 2.1 dargestellt, ausgegeben. Daraufhin wird überprüft, ob 8 Prozesse in einem komplett leeren Prozess-Array angelegt werden können und bei deren Initialisierung immer der nächste freie Slot verwendet wird. Danach wird einer dieser Prozesse wieder entfernt, um zu prüfen, ob `os_exec` einen Prozess in die entstehende Lücke platzieren kann. Anschließend wird die korrekte Initialisierung des Prozess-Arrays für eine Instanz von Programm 9 mit Priorität 10 überprüft. Ein falsch initialisiertes Feld im Prozess-Array wird mit entsprechender Fehlermeldung angezeigt. Schließlich wird überprüft, ob auf dem Prozess-Stack die 33 Nullen für den Laufzeitkontext und das High- und Low-Byte der Progammadresse korrekt geschrieben wurden. Falls die Register falsch initialisiert wurden oder in den darunter liegenden Stack (ISR-Stack) geschrieben wurde, wird eine entsprechende Fehlermeldung ausgegeben, wie in Abbildung 2.2 dargestellt.
Nachdem alle Phasen erfolgreich ausgeführt wurden, zeigt das LCD `„TEST PASSED"` an.

Fehlermeldungen

`„Expected invalid processs"`
`os_exec()` hat nicht `INVALID_PROCESS` zurückgegeben, obwohl entweder `os_processes` voll ist oder eine ungültige ProgramID übergeben wurde.

„Expected valid PID"
`os_exec()` hat `INVALID_PROCESS` zurückgegeben, obwohl in `os_processes` noch freie Slots existieren.

„Not first free slot"
`os_exec()` hat einen neuen Prozess nicht im ersten freien Slot von `os_processes` platziert.

„Invalid PID with gap"
Wenn nur ein Slot in der Mitte des Arrays frei ist, kann `os_exec()` keinen neuen Prozess ausführen und gibt `INVALID_PROCESS` zurück.

„Incorrect PID with gap"
Wenn nur ein Slot in der Mitte des Arrays frei ist, nutzt `os_exec()` diesen nicht, überschreibt einen bereits belegten Slot.

„PID not 0"
Der erzeugte Prozess wurde nicht an den Prozessslot 0 gesetzt.

„Priority not 10"
Der Leerlaufprozess hat nicht die Priorität 10.

„Program pointer invalid"
Der Funktionszeiger des Leerlaufprozesses zeigt nicht auf das Leerlaufprogramm.

„State not READY"
Der Leerlaufprozess wurde nicht auf `OS_PS_READY` gesetzt.

„SP invalid"
Das `SP`-Register zeigt nicht auf die erwartete Speicheradresse.

„Non-zero for register"
Es wurden nicht korrekt 33 Nullen auf den Prozesstack gelegt.

„Invalid hi byte"
Das high-Byte für die Programmadresse ist falsch gesetzt auf dem Stack.

„Invalid lo byte"
Das low-Byte für die Programmadresse ist falsch gesetzt auf dem Stack.

„Written into wrong stack"
Es wurde etwas im darunter liegenden Stack überschrieben.

2.1.2 `os_initScheduler`

Der Unittest `os_initScheduler` überprüft, ob `os_initScheduler()` registrierte Programme korrekt initialisiert. Dazu wird neben `Programm 0`, das für den Idletask benutzt wird, ein Dummy-Programm `Programm 1` erstellt. Anschließend wird `os_initScheduler` aufgerufen und überprüft, ob die ersten beiden Slots des Prozess-Arrays entsprechend initialisiert wurden und die restlichen Slots unbenutzt sind. Im Fehlerfall wird eine Feh-

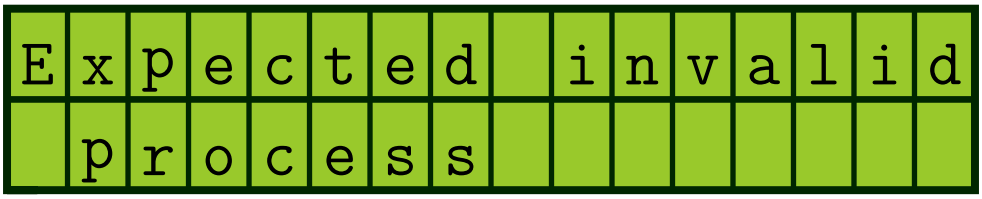

Abbildung 2.1: `os_exec()` liefert nicht `INVALID_PROCESS`

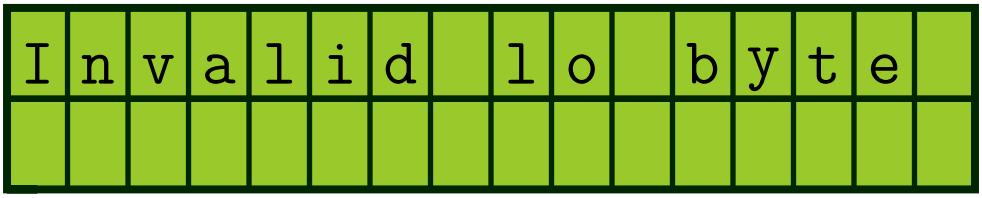

Abbildung 2.2: Low-Byte der Programmadresse nicht korrekt

lermeldung ausgegeben, wie in Abbildung 2.3 dargestellt.
Nachdem alle Phasen erfolgreich ausgeführt wurden, zeigt das LCD „`TEST PASSED`" an.

Fehlermeldungen

„Idle not ready"
Der Leerlaufprozess ist nicht als `OS_PS_READY` markiert.

„Idle not default priority"
Die Priorität des Idle-Prozesses entspricht nicht der `DEFAULT_PRIORITY`.

„Program 1 not started"
Programm 1 wurde nicht gestartet.

„Prog. 1 not default prio."
Die Priorität von Programm 1 entspricht nicht der `DEFAULT_PRIORITY`.

„Other slots not unused"
Die Prozessslots 2 bis `MAX_NUMBER_OF_PROCESSES` sind nicht als `OS_PS_UNUSED` markiert.

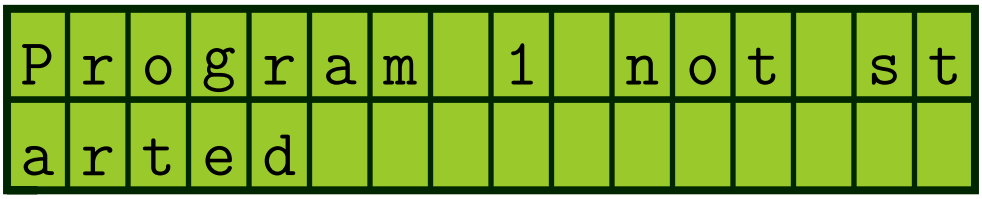

Abbildung 2.3: Programm 1 wurde nicht korrekt initialisiert

2.2 Error

Der Testtask *Error* überprüft, ob die Funktion `os_errorPStr` korrekt implementiert wurde. Dafür durchläuft er 5 Phasen. In den ersten beiden wird überprüft, ob die vier Tasten des Evaluationsboards korrekt genutzt werden können. In den anderen wird dann die Funktionalität von `os_errorPstr` überprüft. Möchte man nur die Funktion der Buttons oder nur die Funktion von `os_errorPstr` überprüfen, kann man dies mit den Defines `PHASE_BUTTONS` und `PHASE_ERROR` einstellen.

Phase 1 In Phase 1 wird kontrolliert, ob eine korrekte Initialisierung der Register `DDRC` und `PORTC` vorliegt. Währenddessen wird keine Interaktion von außerhalb benötigt. Ist die korrekte Initialisierung gegeben, wird dies auf dem LCD angezeigt.

Phase 2 In Phase 2 des Testtasks wird getestet, ob die Eingabe der einzelnen Taster korrekt erkannt wird. Dazu wird der Benutzer aufgefordert den entsprechenden Taster auf dem Evaluationsboard zu drücken und wieder loszulassen, so wie es auf dem LCD gefordert wird.

Phase 3 In Phase 3 wird überprüft, ob die Funktion `os_errorPStr` die Interrupts bei einem Fehler global deaktiviert. Dazu wird `os_errorPStr` mit dem in Abbildung 2.4 dargestellten Text aufgerufen. Die Meldung muss in dieser Form auf dem LCD erscheinen und durch Drücken der Tasten `Enter` und `ESC` quittiert werden. Anschließend wird auf dem LCD das Ergebnis der Prüfung angezeigt. Ist ein Fehler aufgetreten, erscheint auf dem Display die in Abbildung 2.5 dargestellte Meldung.

Phase 4 In Phase 4 wird überprüft, ob der Zustand des *Global Interrupt Enable* Bit im Statusregister nach Quittierung einer Fehlermeldung korrekt wiederhergestellt wird, wenn dieser vor dem Aufruf aktiviert war. Dazu wird `os_errorPstr` mit der Meldung „`Confirm error [GIEB on]"` aufgerufen. Werden die Interrupts durch die Implementierung der Funktion `os_errorPStr` fälschlicherweise nicht reaktiviert, so erscheint hier die Meldung, dass Phase 2 des Tests fehlgeschlagen ist (ähnlich zu der in 2.5 dargestellten Meldung).

Phase 5 In Phase 5 wird überprüft, ob der Zustand des *Global Interrupt Enable* Bit im Statusregister nach Quittierung einer Fehlermeldung korrekt wiederhergestellt wird, wenn dieser vor dem Aufruf deaktiviert war. Dazu wird `os_errorPstr` mit der Meldung „`Confirm error [GIEB off]"` aufgerufen. Werden die Interrupts durch die Implementierung der Funktion `os_errorPStr` fälschlicherweise aktiviert, so erscheint hier eine Fehlermeldung (ähnlich zu der in 2.5 dargestellten Meldung).

Nachdem alle Phasen erfolgreich ausgeführt wurden, zeigt das LCD „`TEST PASSED"` an.

Hinweis

Sollte der Testtask nicht starten, sondern auf dem LCD-Display *Booting* zu lesen sein, liegt meistens ein Fehler in der `os_initScheduler` bzw. der `os_startScheduler` vor. Bitte überprüfen Sie diese nochmals.

Fehlermeldungen

„DDR wrong"
Das `DDRC` Register wurde nicht richtig auf Eingang konfiguriert.

„No pullups"
Die Pullup-Widerstände wurden nicht korrekt für die Buttons konfiguriert.

„waitForInput"
In der Funktion `os_waitForInput()` wurde nicht gewartet, bis ein Taster gedrückt wurde. Falls dieser Fehler erst nach dem Drücken eines Tasters erscheint, kann auch ein Problem in der Funktion `os_getInput()` vorliegen.

„getInput"
Der Rückgabewert der Funktion `os_getInput()` entsprach nicht dem erwarteten Wert.

„waitForNoInput"
In der Funktion `os_waitForNoInput()` wurde nicht gewartet, bis alle Taster losgelassen wurden.

„Missing error"
Der Aufruf oder die Bestätigung der Funktion `os_errorPstr` konnten nicht festgestellt werden.

„Interrupted"
In der Funktion `os_errorPstr` wurden die globalen Interrupts nicht deaktiviert, als die Funktion betreten wurde.

„GIEB falsely off"
Nach dem Aufruf von `os_errorPstr` wurden die globalen Interrupts nicht wieder reaktiviert, obwohl sie zuvor aktiviert waren.

„GIEB falsely on"
Nach dem Aufruf von `os_errorPstr` wurden die globalen Interrupts aktiviert, obwohl sie zuvor deaktiviert waren.

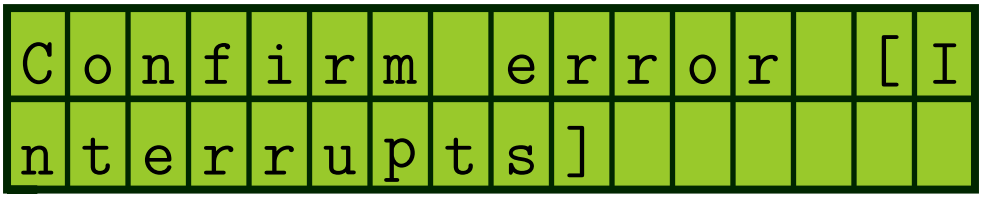

Abbildung 2.4: Confirm Error

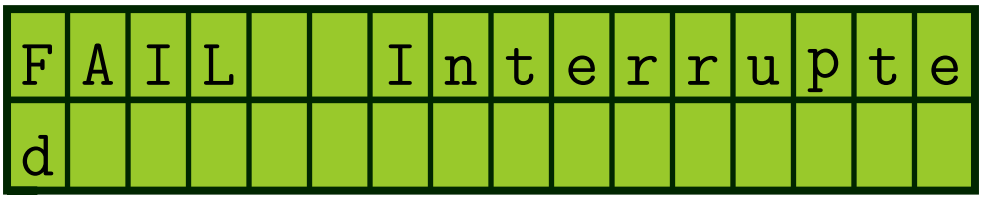

Abbildung 2.5: Phase 3 failed

2.3 Critical

Der Testtask *Critical* überprüft, ob das Betreten und Verlassen von kritischen Bereichen korrekt implementiert wurde. Der Testtask durchläuft dabei vier Phasen.

Phase 1 In der ersten Phase wird kontrolliert, ob durch das Betreten eines kritischen Bereiches nur der Scheduler-Interrupt deaktiviert wird. Dazu muss innerhalb von 15 Sekunden die “Enter”-Taste auf dem Evaluationsboard gedrückt werden. Bei korrekter Implementierung wird die Phase beendet und zu Phase 2 übergegangen. Bei einem Fehler wird dieser auf dem LCD angezeigt und der Testtask angehalten.

Phase 2 Die zweite Phase überprüft, ob Überläufe und Unterläufe des Zählers korrekt mit Fehlern behandelt werden. Dafür müssen zwei Fehler nach der Ausgabe von `„Please confirm Overflow/Underflow"` bestätigt werden. Ist das nicht der Fall, so wird dies auf dem LCD angezeigt.

Phase 3 In der dritten Phase wird getestet, ob der Zustand des GIEB korrekt wiederhergestellt wird. Dazu wird je einmal mit ein- und ausgeschaltetem GIEB eine kritische Sektion betreten und verlassen.

Phase 4 Phase vier testet ob die in `os_exec()` geöffneten kritischen Bereiche in allen Fällen auch wieder geschlossen werden. Ist das der Fall dann wird dies auf dem LCD quittiert, ansonsten tritt die Fehlermeldung `„Crit. Section not closed"` auf.

2.4 Multiple

Der Testtask *Multiple* überprüft, ob von einem Anwendungsprogramm mehrere Instanzen gestartet werden können.
Dazu werden insgesamt sechs Prozesse erstellt, wovon fünf Prozesse dasselbe Anwendungsprogramm ausführen. Bei einer korrekten Implementierung wird fortlaufend `„1; 2a; 2b; 2c; 2d; 2e;"` auf dem LCD ausgegeben (vgl. Abbildung 2.6). Dabei beschreibt die Ziffer das Anwendungsprogramm, welches aktuell ausgeführt wird. Ein angehängter Buchstabe hingegen spiegelt eine Instanz des Anwendungsprogramms wieder. Die Reihenfolge der Zeichenketten ist dabei beliebig, es muss jedoch jede mögliche Ausgabe irgendwann auftreten. Es wird außerdem überprüft, ob der Zustand der Prozesse korrekt gesetzt wird. Wenn alle Instanzen innerhalb von 2 Minuten erfolgreich gestartet wurden, wird `„TEST PASSED"` ausgegeben.

Fehlermeldungen

`„Current proc not running"`
Für den aktuell laufenden Prozess ist nicht der Zustand `OS_PS_RUNNING` im Prozess-Array gesetzt.

`„Other proc is running"`
Ein anderer Prozess außer dem gerade laufenden hat den Zustand `OS_PS_RUNNING` im Prozess-Array gesetzt.

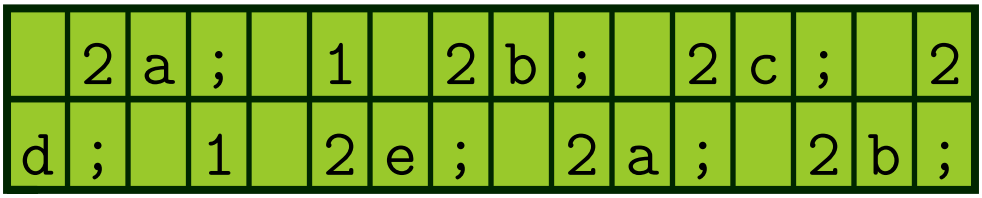

Abbildung 2.6: Multiple

2.5 Resume

Der Testtask *Resume* überprüft, ob Prozesse nach jedem Aufruf des Schedulers erfolgreich fortgeführt werden, d.h. ihr Laufzeitkontext muss korrekt vom Stack geladen und zur Fortführung des Programms verwendet werden.
Hierzu gibt ein Prozess kontinuierlich Buchstaben auf dem LCD aus, während ein zweiter Prozess eine Ziffer ausgibt. Diese Ziffer jedoch wird nur inkrementiert und erneut ausgegeben, nachdem ein dritter Prozess erfolgreich aufgerufen wurde. Demzufolge werden bei einer korrekten Implementierung aufsteigend Zahlen und Buchstaben gemäß der

Abbildung ausgegeben (vgl. Abbildung 2.7). Wenn genügend Zahlen und Buchstaben innerhalb von 3 Minuten korrekt ausgegeben wurden, wird „TEST PASSED" angezeigt.

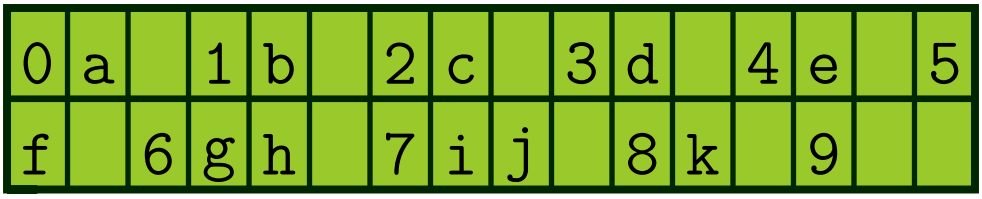

Abbildung 2.7: Resume

2.6 Stack Consistency

Der Testtask *Stack Consistency* überprüft, ob auftretende Inkonsistenzen der Prozessstacks korrekt erkannt werden. Es ist wahrscheinlich, dass dieser Testtask fehlschlägt, wenn bereits der Testtask *Error* nicht erfolgreich gewesen ist.
Für den Test werden zwei Programme automatisch gestartet, welche zuerst die Korrektheit ihres SP überprüfen. Im Anschluss erzeugt das Testprogramm Bitflips im Stackbereich des Hilfsprogrammes nach festgelegten Mustern, und prüft jeweils, ob eine Fehlermeldung bezüglich Stackinkonsistenz vom Scheduler ausgegeben wird. Diese Fehlermeldung von SPOS muss durch gleichzeitiges Drücken der Tasten ENTER und ESC quittiert werden, damit der Test weiterläuft. Der Test erwartet eine XOR-basierte Prüfsumme wie im Dokument beschrieben, die den jeweiligen Stack genau abdeckt. Wenn das Verhalten der Stacküberprüfung nicht mit der Spezifikation übereinstimmt, gibt der Test eine nicht quittierbare Fehlermeldung beginnend mit „FAIL" aus, die den falsch behandelten Fall beschreibt (siehe unten).
Nachdem alle Phasen erfolgreich ausgeführt wurden, zeigt das LCD „TEST PASSED" an.

Fehlermeldungen

„Invalid SP"
Der Stackpointer zeigt nicht auf den Anfang des jeweiligen Prozessstacks.

„Errflag after init"
Es wurde schon während der Initialisierung ein Fehler festgestellt.

„Bit below stack detected"
Es wurde ein Bitflip unterm Stack erkannt, der keine Stackinkonsistenz ist.

„Bit above stack detected"
Es wurde ein Bitflip überm Stack erkannt, der keine Stackinkonsistenz ist.

„Double bitflip detected"
Es wurde eine Stackinkonsistenz erkannt, die aber durch die erwarteten Implementierung nicht erkennbar ist.

„Change not detected"
Es wurde keine Stackinkonsistenz vom Betriebssystem festgestellt.

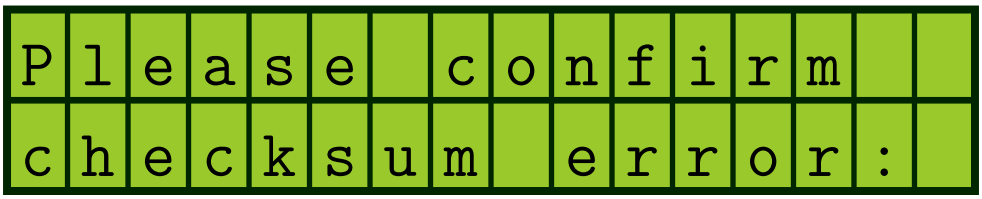

Abbildung 2.8: Stack Consistency

2.7 Scheduling Strategies

Mithilfe des Testtasks *Scheduling Strategies* können alle vorgegebenen Schedulingstrategien überprüft werden. Sollen nur bestimmte Strategien getestet werden, können die anderen dementsprechend in den Defines deaktiviert werden. Dazu werden diverse Prozesse gestartet und das korrekte Scheduling in vier Phasen überprüft:

Phase 1 In der ersten Phase werden die Prozess-IDs der Prozesse in der Reihenfolge ausgegeben, in welcher sie vom Scheduler Rechenzeit zugewiesen bekommen. Bei korrekter Implementierung der Schedulingstrategien ergibt sich beim Durchlauf der ersten Phase des Testtasks folgende Ausgabe, gefolgt von einem „OK":

Even:	„12312312312312312312312312312"
Random: (Seed 1)	„13113333113233313212112213111121"
RoundRobin:	„11222223333333333333333311222223"
InactiveAging:	„13332333231323332331323332331323"
RunToCompletion:	„11111111111111111111111111111111"

Wird von einer Schedulingstrategie Prozess 0 ausgewählt, erscheint der Fehler „Idle returned". Man beachte, dass die Strategien im Codegerüst, sofern nicht geändert, dieses Verhalten aufweisen. Wird ein Fehler erkannt, bleibt die Ausgabe sichtbar und die erste falsche Stelle wird markiert.
Vor dem Aufruf der Random Strategie wird der interne Zustand des Pseudozufallszahlengenerators zurückgesetzt, sodass `rand()` eine bekannte Zahlenfolge zurück gibt und die Ausgabe dieser Strategie somit ebenfalls überprüfbar ist. Hierfür ist es wichtig, dass Sie sich an den Hinweis zur Implementierung halten.

Phase 2 In Phase 2 wird geprüft, ob die Scheduling Strategien den Leerlaufprozess auswählen, wenn kein anderer Prozess ausgeführt werden kann. Es wird ggf. eine Feh-

lermeldung mit der entsprechenden Strategie ausgeben und die weitere Ausführung unterbrochen.

Phase 3 In der dritten Phase wird geprüft, ob alle Prozesse außer dem Leerlaufprozess mindestens ein mal von jeder Strategie ausgewählt werden. Da die Prozesse in diesem Testtask nicht terminieren, ist RunToCompletion von dieser Phase ausgenommen.

Phase 4 Diese Phase testet analog zur vorherigen, allerdings sind hier nicht alle Process Slots belegt.

Phase 5 Diese Phase prüft für alle Strategien, ob der aktuell laufende Prozess erneut ausgewählt wird, wenn nur dieser und der Idle-Prozess den Zustand `OS_PS_READY` haben.

Nachdem alle Phasen erfolgreich ausgeführt wurden, zeigt das LCD „`TEST PASSED`" an.

Fehlermeldungen

„`Idle returned`"
Es wurde unerwartet Prozess 0 ausgewählt.

„S`:` x `not schedulable`"
In der Strategie S wurde Prozessslot x nicht ausgewählt.

„S`: Idle not scheduled`"
In der Strategie S wird der Leerlaufprozess nicht ausgewählt.

„S `Expect` x`, got 0`"
Prozess x ist `READY` und sollte dementsprechend ausgewählt werden, die Strategie S hat aber stattdessen 0 (Leerlaufprozess) zurückgegeben.