

## Praktikum Systemprogrammierung

### Versuch 2

### *Testtaskbeschreibung*

Lehrstuhl Informatik 11 - RWTH Aachen

6. April 2021

Commit: dd0ebb7b

# Inhaltsverzeichnis

<b>2</b>	<b>Testtaskbeschreibung</b>	<b>3</b>
2.1	Unittests . . . . .	3
2.1.1	os_exec . . . . .	3
2.1.2	os_initScheduler . . . . .	5
2.2	Critical . . . . .	6
2.3	Init . . . . .	6
2.3.1	Fehlermeldungen . . . . .	6
2.4	Multiple . . . . .	7
2.5	Resume . . . . .	7
2.6	Stack Consistency . . . . .	8
2.7	Error . . . . .	8
2.7.1	Fehlermeldungen . . . . .	9
2.8	Schedulingstrategien . . . . .	10

---

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 eine ungültige Programm-Id übergeben wird. Falls `os_exec` dies nicht abfängt, wird eine entsprechende Fehlermeldung, wie in Abbildung 2.1 dargestellt, ausgegeben. 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 Programmadresse 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.

#### Fehlermeldungen

##### **Expected invalid processs**

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

##### **PID not 0**

Der erzeugte Prozess wurde nicht an den Prozessslot 0 gesetzt.

##### **Priority not 10**

Der Leerlaufprozess hat nicht die Priorität 10.

**ProgID not 9**

Im Prozessarray `os_processes` hat der erzeugte Prozess nicht die ProgramID 9.

**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.

E	x	p	e	c	t	e	d		i	n	v	a	l	i	d
	p	r	o	c	e	s	s								

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

I	n	v	a	l	i	d		l	o		b	y	t	e	

Abbildung 2.2: Low-Byte der Programmadresse nicht korrekt

### 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 Fehlermeldung ausgegeben, wie in Abbildung 2.3 dargestellt.

#### 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 priority

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.

P	r	o	g	r	a	m		1		n	o	t		s	t
a	r	t	e	d											

Abbildung 2.3: Programm 1 wurde nicht korrekt initialisiert

## 2.2 Critical

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

Der Testtask durchläuft dabei zwei Phasen. 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 zwei 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 Display quittiert, ansonsten tritt die Fehlermeldung „Critical Section not closed“ auf.

## 2.3 Init

Der Testtask *Init* überprüft, ob neue Prozesse bei ihrer Erstellung korrekt initialisiert werden.

Dazu werden insgesamt vier verschiedene Prozesse gestartet, 2 automatisch und 2 manuell. Anschließend wird für jeden Prozess überprüft, ob der jeweilige Stack des Prozesses zutreffend vorbereitet wurde, d.h. ob insbesondere der Laufzeitkontext, also die 32 Prozessorregister, fehlerfrei mit 0 initialisiert wurde.

Auf dem LCD wird eine Tabelle ausgegeben, in der zu jedem der vier Prozesse angegeben wird, ob sein Stack korrekt initialisiert wurde. Ein OK bedeutet, dass der Stack korrekt initialisiert wurde und ein XX bedeutet, dass dies nicht der Fall ist.

Wurden alle Stacks korrekt initialisiert, blinkt auf dem LCD der Text „ALL TESTS PASSED“.

Bitte beachten Sie, dass dieser Test vermutlich fehlschlägt, sollten Sie die Optimierungen ausgeschaltet haben. Wie Sie die Optimierungen einschalten ist dem begleitenden Dokument in Abschnitt 6.3.2 *Probleme bei der Speicherüberwachung* zu entnehmen.

### 2.3.1 Fehlermeldungen

#### `os_exec failed`

`os_exec()` hat `INVALID_PROCESS` zurückgeliefert

1			2			3			4						
OK			OK			XX			OK						

Abbildung 2.4: Ausgabe, falls der Stack von Prozess 3 fehlerhaft initialisiert wurde.

## 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 2.5). 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.

	2	a	;		1		2	b	;		2	c	;		2
d	;		1		2	e	;		2	a	;		2	b	;

Abbildung 2.5: 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 2.6).

0	a		1	b		2	c		3	d		4	e		5
f		6	g	h		7	i	j		8	k		9		

Abbildung 2.6: Resume

## 2.6 Stack Consistency

Der Testtask *Stack Consistency* überprüft, ob auftretende Inkonsistenzen der Prozess-stacks korrekt erkannt werden.

Dazu wird mit Hilfe des Testtasks eine Stackinkonsistenz provoziert, die eine entsprechende Fehlermeldung zur Folge haben soll. Eine Fehlermeldung soll so lange bestehen, bis diese durch gleichzeitiges Drücken der Tasten **Enter** und **ESC** quittiert wird. Wichtig ist, dass durch die Quittierung einer Fehlermeldung nicht der Taskmanager aufgerufen wird. Es ist zu beachten, dass die Stackinkonsistenz bei korrekter Implementierung nicht unmittelbar auftritt, sondern zuerst einige aufsteigende Zahlen ausgegeben werden (vgl. Abbildung 2.7). Nachdem eine Stack-Inkonsistenz aufgetreten ist, verhält sich SPOS undefiniert.

### Fehlermeldungen

#### Consistency check broken

Es wurde keine Stackinkonsistenz vom Betriebssystem festgestellt.

1	;		1	6	9	;		1	7	0	;		1	7	1
;		1	7	2	;		1	7	3	;					

Abbildung 2.7: Stack Consistency

## 2.7 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.

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.

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.

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 2.8 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



## 2 Testtaskbeschreibung

das Ergebnis der Prüfung angezeigt. Ist ein Fehler aufgetreten, erscheint auf dem Display die in Abbildung 2.9 dargestellte Meldung.

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 „GIEB on Confirm error!“ 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 2.9 dargestellten Meldung). 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 „GIEB off Confirm error! " aufgerufen. Werden die Interrupts durch die Implementierung der Funktion `os_errorPstr` fälschlicherweise aktiviert, so erscheint hier eine Fehlermeldung (ähnlich zu der in 2.9 dargestellten Meldung).

Sind alle Phasen erfolgreich durchlaufen, wird auf dem LCD die Meldung ALL TESTS PASSED ausgegeben.

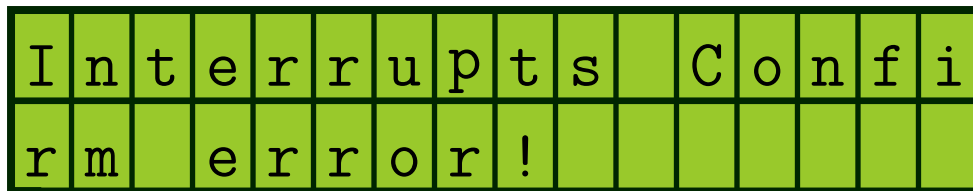


Abbildung 2.8: Confirm Error

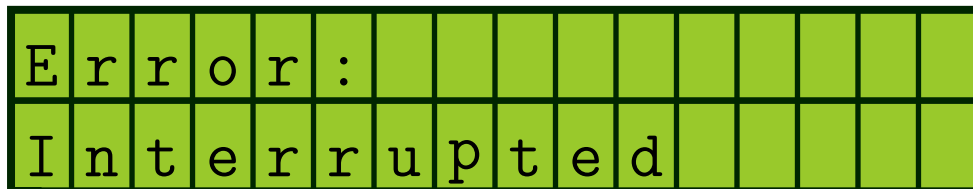


Abbildung 2.9: Phase 1 failed

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.

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.

### 2.7.1 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.

### 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.

## 2.8 Schedulingstrategien

Mithilfe des Testtasks *Schedulingstrategien* können alle vorgegebenen Schedulingstrategien überprüft 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:	12312312312312312312312312312312
Random: (Seed 1)	13113333113233313212112213111121
RoundRobin:	1122223333333333333333331122223
InactiveAging:	13332333231323332331323332331323
RunToCompletion:	11111111111111111111111111111111

## 2 Testtaskbeschreibung

Wird von einer Schedulingstrategie Prozess 0 ausgewählt, erscheint die Meldung „**Not impl. or idle returned**“ und es wird mit der nächsten Strategie fortgefahren. 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 Fehlermeldung 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.

Sollen nur bestimmte Strategien getestet werden, können die anderen dementsprechend in dem Array **strategies** in der Testdatei auskommentiert werden.

### Fehlermeldungen

#### **STRAT: x not schedulable**

In der Strategie **STRAT** wurde Prozessslot x nicht ausgewählt.

#### **STRAT: Idle not scheduled**

In der Strategie **STRAT** wird der Leerlaufprozess nicht ausgewählt.