Aufgabe 2: Prozesse und Scheduling

Prof. Dr. Ronald C. Moore, Fachbereich Informatik, Hochschule Darmstadt

Die Aufgaben, die hier beschrieben werden, sind inspiriert durch Shivakant Mishra, „Simulationsexperimente zu Moderne Betriebssysteme, Tanenbaum“ bzw. „Simulation Exercises for Operating Systems“, © Pearson 2009 (deutsch) und 2015 (englisch).

Die folgende Aufgabe wurde von Prof. Moore für seine Praktika vorbereitet und ist nicht identisch *mit der Aufgabe von Tanenbaum und Mishra!*

Aufgabe 2: Simulationen von Prozessverwaltung

In dieser Aufgabe experimentieren Sie mit Prozessen, die Sie selbst *simulieren*. Das gesamte System wird in Abbildung 1 dargestellt (s. u.). Es geht um die *Implementierung von Prozessen innerhalb eines Betriebssystems.*

Die simulierte CPU

Entwickeln Sie ein Programm, das genau eine CPU simuliert. Als Folge kann der Simulations-Manager zu jeder gegebenen Zeit nur ein simulierter Prozess ausführen; nur ein simulierter Prozess ist „aktiv“. Die Simulation kann von der Benutzerin gesteuert werden.

Die simulierte CPU hat zwei Register – eine Ganzzahl (Integer) und einen Programmzähler. (Zusätzlich hat die Simulation eine Uhr). Simulierte Prozesse können angehalten und ausgewechselt werden, sodass andere Prozesse simuliert werden können. Wenn ein Prozesswechsel stattfindet, müssen die zwei Register gespeichert werden, sodass deren Inhalte später wiederhergestellt werden können (wenn der angehaltene Prozess wieder zum Laufen gebracht wird).

Die simulierte CPU hat ihren eigenen Befehlssatz, der unten beschrieben wird.

Jeder simulierte Prozess führt ein Programm aus. Die Programme werden aus Linux-Dateien gelesen und im Befehlsspeicher gespeichert. Wenn ein neuer Prozess simuliert wird, wird zuerst seine Programm-Datei gelesen und gespeichert werden.

Am Anfang der Simulation wird das erste Programm aus einer Datei namens „init“ geladen und ausgeführt – dieser Dateiname ist fest vordefiniert! Dieser Prozess wird in der Regel andere Prozesse ins Leben rufen, die wiederum noch mehr Prozesse ins Leben rufen (können).

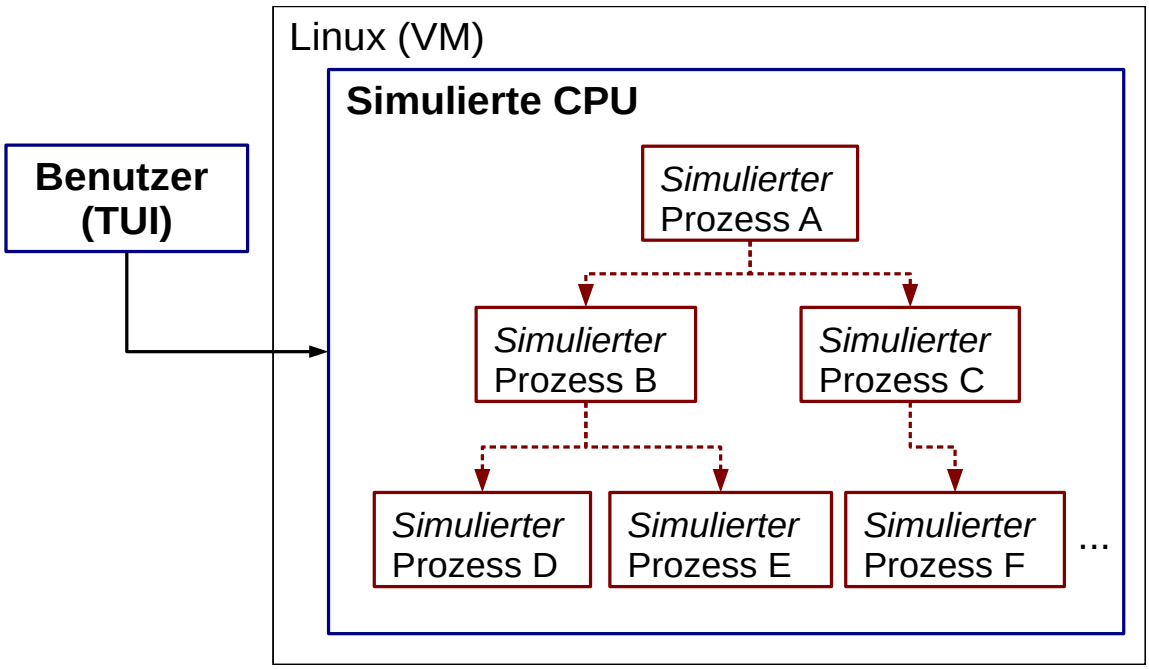

Abbildung 1 – System Architektur: Rote Kästen sind *simulierte* Prozesse, die von dem Prozessmanager *simuliert* werden. Die sind nicht mit echten Linux-Prozesse, die etwa mit `fork()` ins Leben gerufen werden, zu verwechseln.

Es gibt ein „Block“ Befehl, womit ein Prozess stoppt, bis die Benutzerin sie wieder freigibt. Dieser Befehl ist da, um sowohl Ein- als auch Ausgabe zu simulieren.

Benutzer ↔ Simulation Schnittstelle

Es gibt folgende Arten von Befehlen, die die Benutzerin eingeben darf:

1. **`S` *`n`*** oder **`Step` *`n`*** – *n* Takte werde simuliert. Falls *n* nicht angegeben wird, ist *n* = 1 (der Defaultwert von *n* ist 1). Hinweis: Es kann sein, dass ein Prozess während der Simulation blockiert wird. Was danach passiert, hängt von der Simulation ab. Die Uhr wird allerdings nie blockiert – die Zeit schreitet immer weiter fort.

2. **`U`** oder **`Unblock`** – der erste blockierte simulierte Prozess wird „entblockiert“, d. h., der älteste Prozess in der Warteschlange blockierter simulierter Prozesse wird aus der Warteschlange entfernt und *kann* nun ausgeführt werden – es *darf* ein Prozesswechsel stattfinden.

3. **`P`** oder **`Print`**: Ausgabe des aktuellen Zustands des Systems.

4. **`Q`** oder **`Quit`:** Ausgabe der durchschnittlichen simulierten Durchlaufzeit (Takte per Prozess) und Beendigung des Systems.

Der Befehlssatz der simulierter CPU

Der Befehlssatz der simulierten CPU ist wie folgt:

1) **`S`** ***n*:** Setze den Wert des Integerregisters auf *n*, wo *n* eine Ganzzahl ist.
2) A n: Füge dem Wert des Integerregisters n hinzu, wo n eine Ganzzahl ist
3) **`D`** ***n***: Subtrahiere *n* vom Wert des Integerregisters, wo *n* eine Ganzzahl ist.
4) **`B`**: Blocke diesen simulierten Prozess.
5) **`E`:** Beende diesen simulierten Prozess.
6) **`R`** ***Dateiname***: Erstelle einen neuen simulierten Prozess. Der neue simulierte Prozess führt das Programm von Datei ***Dateiname*** aus. D. h. die Datei wird gelesen, deren Inhalt wird im Befehlsspeicher gespeichert, und ein neuer Prozess wird in der Liste der ausführungsbereiten Prozesse hierfür installiert. Es wird **nicht** gewartet, bis der neue Prozess fertig ist – der Aufrufer läuft weiter. Genauer gesagt, beide Prozesse (Aufrufer und Aufgerufener) müssen die simulierte CPU nun teilen. Welche der zwei Prozesse zuerst ausgeführt wird, hängt vom Scheduling-Algorithmus ab.

Sie sollen Ihre eigenen Test-Programme schreiben!

Ein Beispiel-Programm:

```
S 0
A 2
R file_a
A 60
B
R file_b
D 20
B
R file_c
E
```

Dieses Programm soll drei andere Programme ausführen (`file_a, file_b` und `file_c`) und die Zahl 42 berechnen. Dabei wird der Prozess zweimal blockiert, d. h. der Prozess wartet zweimal, bis der **U**-Befehl vom Kommandant kommt (s. o.).

Prozesstabelle und andere Datenstrukturen

Programmieren Sie geeignete Datenstrukturen, um die simulierte CPU zwischen den simulierten Prozessen zu teilen, d. h. Prozesse anhalten und später wieder „aufwecken“ zu können.

Sie können die Datenstrukturen benennen und definieren, wie Sie wollen. Allerdings muss es einen Datensatz pro Prozess geben. Dabei muss mindestens der Zustand der zwei Register (Integerregister und Programmzähler) gespeichert werden. Wir wollen auch wissen, wie viele CPU-Zyklen jeder Prozess bekommen hat, und wie viele Zeit insgesamt verwendet wurde. Sie werden wahrscheinlich zwei Warteschlangen brauchen: Eine für blockierte und eine für bereite (nicht blockierte) Prozesse. Sie sollen auch jedem Prozess eine eindeutige ID geben, wobei der erste Prozess ID 0 hat.

Ausgabe der Simulation

Sie entscheiden, wie die Ausgabe der Simulation auszusehen hat. Die Ausgabe soll aussagekräftig sein und alle Fragen beantworten, die die Simulationsexperimente aufwerfen (s. u.)!

Hier ein **Beispiel** Ausgabe:

```
**************************************************************
The current system state is as follows:
**************************************************************
CURRENT TIME: 69
RUNNING PROCESS:
pid  ppid priority   value  start time     CPU time used so far
2    1    0          42     35             7
BLOCKED PROCESSES:
pid  ppid priority   value  start time     CPU time used so far
1    0    0          21     30             5
4    2    0          19     39             4
  ...
PROCESSES READY TO EXECUTE:
pid  ppid priority   value  start time     CPU time used so far
3    0    0          32     20             12
  ...
**************************************************************
```

In diesem Beispiel gibt es 1 Prozess, der gerade läuft -- der „Running Process“ mit ID 2 und Eltern-ID 1. Der hat Priorität 0, sein Register hat den Wert 42, der startete bei Zeit 35, und hat bis jetzt 7 Takte Zeit verwendet.

Weiter: Es gibt zwei Prozesse, die blockiert sind, mit IDs 1 und 4, und einen Prozess, der bereit ist, d. h. nicht blockiert ist. Der hat ID 3.

Simulationsexperimente

Programmieren Sie Ihren Simulator so, dass es leicht ist, verschiedene Scheduling Algorithmen zu testen. Programmieren Sie mindestens zwei Scheduling Algorithmen.

Dabei soll zuerst ein einfaches Scheduling (ohne Quantum aber mit Unterbrechungen) programmiert werden. Jeder Prozess läuft, bis er blockiert wird (durch ein „B“ CPU-Befehl), oder ein anderer Prozess „entblockiert“ wird (durch ein „U“ Benutzer-Eingabe). Nur dann dürfen andere Prozesse laufen!

Danach schreiben Sie ein „Round-Robin“ Scheduling – ein Prozess wird zusätzlich angehalten, wenn er ein von Ihnen definiertes „Quantum“ (maximale Anzahl von Befehlen bzw. längste erlaubte Zeit) gelaufen ist.

Wenn Sie Zeit haben, dürfen Sie auch mit Priorität-Scheduling bzw. „Completely Fair“ Scheduling experimentieren.

Messen Sie das Verhalten Ihrer Test-Programme mit den verschiedenen Scheduling Algorithmen und halten Sie Ihre Ergebnisse fest, sodass Sie sagen können, ob bzw. wie die verschiedene Algorithmen sich anders verhalten.