Technische Universität Berlin Fachgebiet Komplexe und Verteilte IT-Systeme Sommersemester 2017	**Aufgabenblatt 3** zu – Systemprogrammierung – Prof. Dr. Odej Kao, Dr. Peter Janacik, Julian Lindemeier, Fabian Hofmann
Abgabetermin:	[1] – 04.06.2017 [2] – 11.06.2017

Aufgabe 3.1: Scheduling-Handsimulation (2 Punkte) (Theorie[1])

Es werden in einem Ein-Prozessor-System Prozesse wie in Abbildung 1 beschrieben gestartet:

Prozess	A	B	C	D	E	F
Ankunftszeitpunkt	0	1	4	6	8	9
Dauer	5	1	10	3	7	4
Priorität	3	1	4	2	6	5

Abbildung 1: Prozesse eines Systems mit einer CPU und einem Thread

a) Simulieren Sie folgende Scheduling-Verfahren für die Prozesse aus Abbildung 1:

- LCFS-NP,
- LCFS-P,
- HRRN,
- Prio-P,
- MLF mit $\tau_i = 2^i \ (i = 0, 1, \dots)$.

Geben Sie für *jeden* Zeitpunkt den Inhalt der Warteschlange und den Prozess auf der CPU an.

Die Lösung soll in Form der dargestellten Tabelle abgegeben werden, wobei anzumerken ist, dass für Multilevel-Feedback mehrere Warteschlangen benötigt werden:

Zeit	0	1	2	3	…	28	29
CPU	A	…	…	…	…	…	…
Warteschlange		…	…	…	…	…	…
		…	…	…	…	…	…

b) Berechnen Sie für *jedes* der in a) verwendete Verfahren

- die Warte- und Antwortzeit *jedes* Prozesses sowie
- die mittlere Warte- und Antwortzeit des gesamten Systems.

Aufgabe 3.2: Scheduling-Verfahren (Tafelübung)

Benennen Sie die aus der Vorlesung bekannten Scheduling-Verfahren und ordnen Sie diese nach

a) Strategiealternativen:

- ohne/mit Verdrängung,
- ohne/mit Prioritäten und
- unabhängig/abhängig von der Bedienzeit.

b) Betriebszielen:

- Effizienz/Durchsatz,
- Antwortzeit und
- Fairness.

Begründen Sie Ihre Entscheidungen für die Betriebsziele!

Aufgabe 3.3: Scheduling-Handsimulation (Tafelübung)

Es werden in einem Ein-Prozessor-System Prozesse wie in Abbildung 2 beschrieben gestartet:

Prozess	A	B	C	D
Ankunftszeitpunkt	0	3	4	8
Dauer	5	3	8	4
Priorität	4	1	2	3

Abbildung 2: Prozesse eines Systems mit einer CPU und einem Thread

a) Simulieren Sie folgende Schedulingverfahren für die Prozesse aus Abbildung 2:

- FCFS,
- PRIO-NP,
- SRTN,
- RR mit $\tau = 2$,
- MLF mit $\tau_i = 2^i$ $(i = 0, 1, \dots)$.

b) Berechnen Sie für jedes der in a) verwendeten Verfahren

- die Warte- und Antwortzeit *jedes* Prozesses sowie
- die mittlere Warte- und Antwortzeit des gesamten Systems.

Aufgabe 3.4: Prozessorausnutzung (Tafelübung)

Die Prozessorausnutzung ρ sei als Quotient aus der minimal erforderlichen und der tatsächlich benötigten Zeit zur Ausführung anstehender Prozesse definiert. Dabei soll die Laufzeit eines Prozesses T Zeiteinheiten betragen und ein Prozesswechsel S Zeiteinheiten kosten (es gilt: $S \ll T$).

a) Geben Sie eine alternative Formel zur Berechnung der Prozessorausnutzung für das Round-Robin-Verfahren unter Verwendung der Zeitscheibenlänge τ und der Prozesszahl n an.

b) Berechnen Sie anhand der Formel aus a) die Grenzwerte für folgende Fälle:

- $\tau \to 0$,
- $\tau = S$ und
- $\tau \to \infty$.

c) Stellen Sie die Abhängigkeit von Effizienz und Zeitscheibenlänge grafisch dar.

Aufgabe 3.5: Thread-Scheduler (3 Punkte) (Praxis^2)

In dieser Aufgabe sollen verschiedene Schedulervarianten implementiert werden, um die Ausführungsreihenfolge von Threads zu steuern. Dabei gibt es zwei Arten von Threads: Eine Anzahl von Taskthreads, deren Spezifikationen (Name, Farbe, Dauer, Ankunftszeit) über eine JSON-Datei festgelegt werden und einen Schedulerthread, der je nach Schedulingverfahren entscheidet, in welcher Reihenfolge die Tasks ausgeführt werden. Hierbei wird angenommen, dass Tasks niemals zum selben Zeitpunkt ankommen, um die Aufgabe nahe an der Tafelübung und den Theoriehausaufgaben zu orientieren.

Implementieren Sie die folgenden Scheduler:

a) Einen First-Come-First-Serve Scheduler (FCFS),

b) einen Shortest-Job-Next Scheduler (SJN),

c) einen Round-Robin Scheduler (RR) mit einem Quantum von $\tau = 2$,

d) einen Multilevel-Feedback Scheduler (MLF) mit folgenden Eigenschaften:

- Die Zeitscheiben des i.ten Levels haben eine Größe von $\tau_i = 2^i$ $(i = 0, 1, \dots)$
- Die Anzahl der Level (`num_levels`) wird an die zu implementierende Funktion übergeben, jedoch ist hier wichtig, dass die Warteschlange des untersten Levels nach dem FCFS-Verfahren arbeitet.
- Am Beispiel für `num_levels = 5`:
  - Queue 1: `time_step = 1`
  - Queue 2: `time_step = 2`
  - Queue 3: `time_step = 4`
  - Queue 4: `time_step = 8`
  - Queue 5: FCFS

Hinweise:

- **Funktionsskelette:** Die Funktionsskelette für die zu implementierenden Scheduler sind in dem Vorgabenverzeichnis im *Scheduling*-Ordner in den jeweiligen .c- und .h-Dateien. Die Vorgabe ruft jeden Scheduler über die vorgegebene Funktion auf, weswegen diese die volle Funktionalität des Schedulers bieten muss. Selbstverständlich können weitere Hilfsfunktionen sowie .c und .h Dateien hinzugefügt werden, aber Sie müssen - um die Korrektur zu erleichtern - im Ordner *Scheduling* liegen.

- **Datenstrukturen:** Für die Scheduler werden Sie weitere Datenstrukturen benötigen. Bevor Sie diese erstellen, überlegen Sie sich zunächst, ob sich diese und die dazu gehörigen Funktionen so konstruieren lassen, dass die Datenstrukturen mit neuen Funktionen für alle Scheduler verwendet werden können.

- **Bereitgestellte Funktionen und Datenstrukturen:** In dem bereitgestellten Code finden sich Funktionen, die sich für Ihre Implementierung als hilfreich erweisen könnten. Es ist daher empfehlenswert, sich die Kommentare der Vorgabe gründlich durchzulesen!

- **Beispiel-Scheduler:** Um Ihnen eine kleine Einleitung in die bereitgestellten Funktionen und einen exemplarischen Aufbau der zu implementierenden Scheduler-Funktion zu bieten, gibt es im *Scheduling*-Ordner ein kleines Beispiel namens `Monkey.c`. Um diesen ebenfalls auszuführen, muss jedoch in der `Main.c` der entsprechende Funktionsaufruf entkommentiert werden.

- **Kompilierung:** Für diese Aufgabe werden Ihnen zwei Möglichkeiten geboten, Ihren Code zu kompilieren:

  - **CMake:** CMakeFiles ermöglichen die Erzeugung von Makefiles mit dem Programm `cmake`, sowie erleichtern einigen IDEs die Verwaltung eines Projektes. Um das `makefile` zu erzeugen, muss lediglich im Ordner `cmake CMakeLists.txt` ausgeführt werden. Anschließend kann mit dem üblichen Befehl `make` Ihr Code kompiliert werden.
    Wenn Sie weitere .c Dateien erstellen, müssen diese auch in der Datei `CMakeLists.txt` zu den Quellen hinzugefügt werden. Nach dem Verändern der CMakeFile müssen Sie den oberen Befehl abermals ausführen, um das Makefile zu updaten.
  - **Shell-Skript:** Als Alternative zu cmake erfolgt hier die Kompilierung mit dem Ausführen eines Shell-Skriptes (`./compile.sh`). Das Skript arbeitet für die Quelldateien mit Wildcards, weswegen neue Quellen nicht explizit angegeben werden müssen. Natürlich können Sie auch den gcc-Aufruf aus dem Shell-Skript kopieren und ihn über das Terminal ausführen, wenn keine der beiden gestellten Möglichkeiten zum Kompilieren genutzt werden sollen.

- **Ausführen und Testen:** Nach dem Kompilieren können Sie Ihre Implementierungen testen, indem Sie diese mit einer der bereits vorgebenen JSON-Dateien als Argument ausführen. Diese enthalten die Spezifikationen mehrerer Tasks sowie die Anzahl der Warteschlangen in MLF und können beliebig verändert werden, um beliebige Szenarien zu testen. Dabei ist zu empfehlen, erst einmal eine Handsimulation für die verwendeten Tasks durchzuführen und dann mit der Ausgabe des Programmes zu vergleichen. Um die Ausgabe des Programmes zu beschleunigen, können Sie das Programm mit der Flag `NO_FANCY` kompilieren, welches standardmäßig in der CMakefile und Shell-Skript kommentiert wurde.
  Sie können JSON-Dateien und die erwarteten Ausgaben mit Kommilitonen austauschen, um Zeit beim Testen zu sparen.

- **Abgabe:** Für die Abgabe sind nur veränderte und neue Dateien im Ordner *Scheduling* relevant.