Technische Universität Berlin Fachgebiet Komplexe und Verteilte IT-Systeme Sommersemester 2019	**Aufgabenblatt 3** zu – Systemprogrammierung – Prof. Dr. Odej Kao
Abgabetermin:	[1] – 9.6.2018 23:55 Uhr [2] – 9.6.2018 23:55 Uhr

Aufgabe 3.1: 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 (BZ).

b) Betriebszielen:

- Effizienz/Durchsatz,
- Antwortzeit und
- Fairness.

Begründen Sie Ihre Entscheidungen für die Betriebsziele!

Aufgabe 3.2: Scheduling-Handsimulation (Tafelübung)

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

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

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

a) Simulieren Sie folgende Schedulingverfahren für die Prozesse aus Abbildung 1 unter Verwendung des SysprogInteract Tools (https://citlab.github.io/SysprogInteract/):

- FCFS,
- PRIO-NP,
- SRTN,
- RR mit $\tau = 2$.

b) Simulieren Sie das MLF Schedulingverfahren für die Prozesse aus Abbildung 1 an der Tafel mit $\tau_i = 2^i \ (i = 0, 1, \dots)$.

c) Berechnen Sie für jedes der verwendeten Verfahren

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

Aufgabe 3.3: 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 3.4: Periodische Prozesse (Tafelübung)

Auf dem Bordcomputer eines Autos sollen Prozesse periodisch ausgeführt werden. Dabei aggregiert Prozess A Sensordaten über die Umdrehungszahl der Räder. Diese Datensätze werden vom ABS (Prozess B) verarbeitet. Prozess C überprüft Kollisionssensoren und löst, wenn nötig, den Airbag aus.

Die Startzeitpunkte, Perioden und die von den Prozessen auf dem Bordcomputer benötigte Laufzeit können Sie der Tabelle 1 entnehmen.

Prozess	Startzeitpunkt	Dauer	Periode
A	0	2	6
B	1	3	12
C	2	1	4

Tabelle 1: Prozess-Charakteristika

Außerdem gilt, dass die Frist (Deadline) eines Prozesses seine Periode ist.

a) Existiert für diese Prozesse ein zulässiger Schedule? Wie könnte dieser aussehen? Wie groß ist die Hyperperiode?

b) Was passiert, wenn zu den Prozessen ein Weiterer (D: 4, 2, 6) hinzugefügt wird?

c) Was ist der Unterschied zwischen Hard- und Softlimits bei Echtzeitsystemen? Finden Sie jeweils Beispiele.

Aufgabe 3.5: Scheduling-Theorie (1 Punkt) (Theorie[1])

a) Erklären sie den Begriff *Verdrängung*. **(0,1 Punkte)**

b) Zwischen welchen zwei Schedulingzielen bildet das HRRN-Verfahren einen Kompromiss? **(0,2 Punkte)**

c) Erklären sie das Prinzip der *Prioritätsvererbung*. Was kann ohne sie passieren? **(0,2 Punkte)**

d) Was sind die Unterschiede zwischen *Online-* und *Offline-Scheduling*? Gehen sie dabei auch auf die benötigten Voraussetzung für beide ein. **(0,2 Punkte)**

e) Was sind die Unterschiede und Gemeinsamkeiten von *Hard-* und *Soft-real-time-Systems*. Nennen sie jeweils ein Beispiel. **(0,3 Punkte)**

Aufgabe 3.6: Scheduling-Handsimulation (1 Punkt) (Theorie^2)

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

Prozess	A	B	C	D	E
Ankunftszeitpunkt	0	2	4	6	9
Dauer	5	3	6	2	7

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

a) Simulieren Sie folgende Scheduling-Verfahren für die Prozesse aus Abbildung 2:

- LCFS-PR,
- HRRN,
- 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	...	26	27
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 3.7: Scheduler (3 Punkte) (Praxis^2)

In dieser Aufgabe sollen verschiedene Schedulervarianten implementiert werden.

a) Implementieren Sie einen Scheduler, der nach dem Shortest Job Next(SJN)-Verfahren (nicht präemptiv) arbeitet. (0,5 Punkte)

b) Implementieren Sie je einen Scheduler, der nach dem Last Come First Serve(LCFS)-Verfahren mit Verdrängung und einen Scheduler nach dem Shortest Remaining Time Next(SRTN)-Verfahren mit Verdrängung arbeitet. (1 Punkt)

c) Implementieren Sie je einen Scheduler, der nach dem Round Robin(RR)-Verfahren arbeitet (die Größe der Zeitscheibe wird bei init_RR(int time_step) übergeben mit time_step > 0) und einen Multilevel Feedback(MLF)-Scheduler mit folgenden Eigenschaften: (1,5 Punkte)

- Die init Funktion erwartet die zwei Parameter time_step > 0 (Zeitscheibe der ersten Queue) und num_queues > 0 (Anzahl der Queues).
- num_queues beinhaltet jeweils n-1 RR-Queues inklusive einer FIFO Queue. Bsp: num_queue = 4 bedeutet 3 RR-Queues und 1 FIFO Queue.
- Die jeweilige Zeitscheibe der Round Robin Queues ergibt sich folgendermaßen:

$$zeitscheibe\_fuer(1) = time\_step$$
$$zeitscheibe\_fuer(n) = zeitscheibe\_fuer(n-1) * 2$$

  mit $2 \leq n \leq num\_queues$.
  Konkretes Beispiel für time_step = 2 und num_queues = 4:
  - Queue 1: time step = 2
  - Queue 2: time step = 4
  - Queue 3: time step = 8
  - Queue 4: FIFO
- Der Scheduler soll auch bei neu ankommenden Tasks präemptiv arbeiten: Wenn in einer höheren Ebene(Queue) ein Prozess bereit wird, wird der laufende Prozess verdrängt und der Zustand seiner Ebene (Queue) eingefroren. Die angebrochene Zeitscheibe des Prozesses wird bei der nächsten Auswahl dieser Ebene(Queue) fortgesetzt. Tasks aus der FIFO Queue werden **nicht** verdrängt.
  Das heißt für unser konkretes Beispiel, dass wenn Task A aus Queue 2 im zweiten time_step ist und Task B ankommt, wird Task A zurück an den Anfang der Queue 2 gesteckt (mit dem Vermerk, dass Task A schon zwei time_steps genutzt hat). Solange Task B und möglicherweise auch weitere Tasks in höher priorisierten Queues liegen, muss Task A nun auf die weitere Ausführung warten. Liegt kein Tasks mehr vor Task A, wird Task A unter Berücksichtigung der schon genutzten time_steps in die CPU gesteckt. Somit bleiben Task A noch 2 weitere time_steps in der CPU, bevor er in die Queue 3 präemptiv gesteckt wird (falls Task A noch Ausführungszeit benötigt).

Die Scheduler sollen dabei als Interface folgende Funktionen anbieten:

- `int init_XX()`
  Wird zur Initialisierung des jeweiligen Schedulers aufgerufen. Hier können Sie benötigte Datenstrukturen initialisieren. Die Funktion gibt 0 zurück, wenn die Initialisierung erfolgreich abgeschlossen wurde.
- `void free_XX()`
  Wird aufgerufen, um den evtl. allokierten Speicher Ihrer Datenstrukturen wieder freizugeben.
- `void arrive_XX(int id, int length)`
  Wird aufgerufen, um dem Scheduler einen neuen Prozess zu übergeben. Es wird für jeden Prozess eine Kennung id ≥ 0 und die Gesamtlaufzeit in Zeiteinheiten > 0 übergeben. Prozesse kommen dabei immer am Anfang einer Zeiteinheit an, d.h. noch bevor `tick_XX()` aufgerufen wird.
  Sie können davon ausgehen, dass maximal ein Prozess pro tick ankommen wird.

- `def_task *tick_XX()`
  Für jede vergangene Zeiteinheit wird einmal tick aufgerufen. Tick aktualisiert Datenstrukturen die für die Verwaltung der Prozesse im Scheduler wichtig sind und tauscht bei Bedarf den laufenden Prozess in der CPU aus (mithilfe von switch_task(), siehe dazu task.h).
  Tick gibt den Prozess zurück, der in der laufenden Zeiteinheit in der CPU war.
- `void finish_XX()` (optional)
  Wird aufgerufen, wenn der laufende Prozess retired", also beendet ist. Finish aktualisiert wie tick Datenstrukturen die für die Verwaltung der Prozesse im Scheduler wichtig sind und/oder tauscht den laufenden Prozess in der CPU aus(mithilfe von switch_task(), siehe dazu task.h). Die Implementierung ist optional, da wir diese Funktion nicht testen werden. Das heißt dass finish nur intern von Ihnen aufgerufen werden kann, aber nie durch die main. Finish wird aber dabei helfen, Ihren Code möglichst modular zu halten.

Im Scheduler sollen die Prozesse verwaltet und zu passenden Zeitpunkten aus `arrive()`, `tick()` oder `finish()` heraus Prozesswechsel durchgeführt werden. **Der Task der zuletzt verstrichenen Zeiteinheit soll in der Variable def_task *running_task (siehe include/task.h) bis zum nächsten Aufruf von arrive_XX oder tick_XX (da beide Funktionen die nächste Zeiteinheit einleiten) gespeichert werden. gespeichert werden. Falls kein Prozess läuft und keiner in der Queue auf die Ausführung wartet, soll running_task == NULL gelten.** Um einen Prozesswechsel durchzuführen steht ihnen folgende Funktion zur Verfügung:

- `def_task *switch_task( def_task *task )`
  Wechselt zum übergebenen Prozess (passt die Variable running_task an). Der Return-Wert ist der verdrängte Prozess für eine weitere mögliche Berücksichtigung im Scheduler oder andere Verwaltungsfunktionen.

Zusätzlich wird Ihnen eine Queue-Implementierung vorgegeben, die Sie als Scheduling-Queue zur Verwaltung der Tasks benutzen können. Die Beschreibung der einzelnen Funktionen finden Sie in Queue.h im Ordner include.

Hinweise:

- **Beschreibungen der Funktionen:** Nähere Beschreibungen der einzelnen Funktionen finden Sie auch in den .h-Dateien im Ordner include.
- **Ankommen von Tasks:** Beachten Sie, dass immer zuerst arrive und dann tick ausgeführt werden soll. Beispiel: task1 kommt bei Zeiteinheit 0 an, task2 bei Zeiteinheit 2. Dann ist die Reihenfolge: arrive(task1), tick(), tick(), arrive(task2), tick(), ...
- **Simulation** In der main-Funktion finden Sie einen möglichen Test Ihrer Implementierung durch uns. Ein ausführbares Programm working_example mit der erwarteten Ausgabe wurde im Ordner src hinterlegt. So lernen Sie die von uns erwartete Aufrufsreihenfolge der Schedulerfunktionen anhand des Beispiels aus der Vorlesung 3 auf Seite 12. Weitere Tests können Sie aus diesem Arbeitsblatt, der Vorlesung oder dem Internet entnehmen.
- **Wichtig ist, dass in jedem Fall der Task der letzten verstrichenen Zeiteinheit bis zum nächsten Aufruf von arrive_XX oder tick_XX (da beide Funktionen die nächste Zeiteinheit einleiten) in der Variable running_task hinterlegt ist.**
- **Das Abgabearchiv** soll die Ordner *src* und *include* beinhalten.
- **Vorgaben:** Bitte ändern Sie bestehende Datenstrukturen, Funktionsnamen, ... nicht. Eine Missachtung kann zu Punktabzug führen. Gerne können Sie weitere Hilfsfunktionen oder Datenstrukturen definieren, die Ihnen die Implementierung leichter gestalten.

- **Makefile:** Bitte verwenden Sie für diese Aufgabe das Makefile aus der Vorgabe. Führen Sie hierzu im Unterordner *src* der Vorgabe *make* aus. Durch den Befehl *make clean* werden kompilierte Dateien gelöscht. Gerne können Sie das Makefile um weitere Flags erweitern oder anderweitig anpassen. Ihr Programm sollte aber mit dem vorgegebenen Makefile weiterhin compilierbar bleiben.
- **Dynamischer Speicher:** Um zu evaluieren ob der gesamte vom Scheduler allokierte Speicher wieder freigegeben wurde, empfiehlt sich das Kommandozeilen Werkzeug **valgrind**[1] (unter linux über apt-get install valgrind). **Memory leaks führen zu Punktabzug.**

[1] http://valgrind.org/