Lehrstuhl für Connected Mobility
TUM School of Computation, Information and Technology
Technische Universität München

Grundlagen - Betriebssysteme und Systemsoftware

IN0009, WiSe 2023/24

Übungsblatt 4

13. November 2023 – 17. November 2023

Hinweis: Mit * gekennzeichnete Teilaufgaben sind ohne Lösung vorhergehender Teilaufgaben lösbar.

Aufgabe 1 Prozessverwaltung und Scheduling

Gegeben seien zwei Prozesse P_1 und P_2 mit einer in der folgenden Tabelle vorgegebenen Anzahl an Kernel-Level-Threads K_i und User-Level-Threads U_j. Die folgende Tabelle gibt sowohl die Startzeit als auch die Rechenzeit für alle Prozesse und Threads an.

Prozess	*Thread*	*Startzeit*	*Rechenzeit*
P_1	K_1	0	13
P_1	K_2	3	3
P_1	K_3	20	2
P_2	U_1	2	8
P_2	U_2	9	5
P_2	U_3	12	4

Der **Kernel-Scheduler** arbeitet nach dem **Round-Robin** Verfahren (Zeitscheiben), wobei eine Zeitscheibe **fünf Einheiten** beträgt (Aktivitäten des Schedulers oder Dispatchers nicht mitgerechnet). Der Kernel-Scheduler behandelt **alle** von ihm verwalteten Prozesse und Threads gleich. Sie werden in der Reihenfolge ihrer Startzeit abgearbeitet.

Jedes Mal, wenn der Kernel-Scheduler aktiv wird, benötigt er **eine Zeiteinheit**. Muss zusätzlich ein Prozess-Kontextwechsel durchgeführt werden, kostet das **eine zweite Zeiteinheit**. Beachten Sie, dass die initiale Aktivität des Kernel-Schedulers im Diagramm unten nicht berücksichtigt wird und der erste Prozess zum Zeitpunkt 0 direkt mit der Ausführung beginnt.

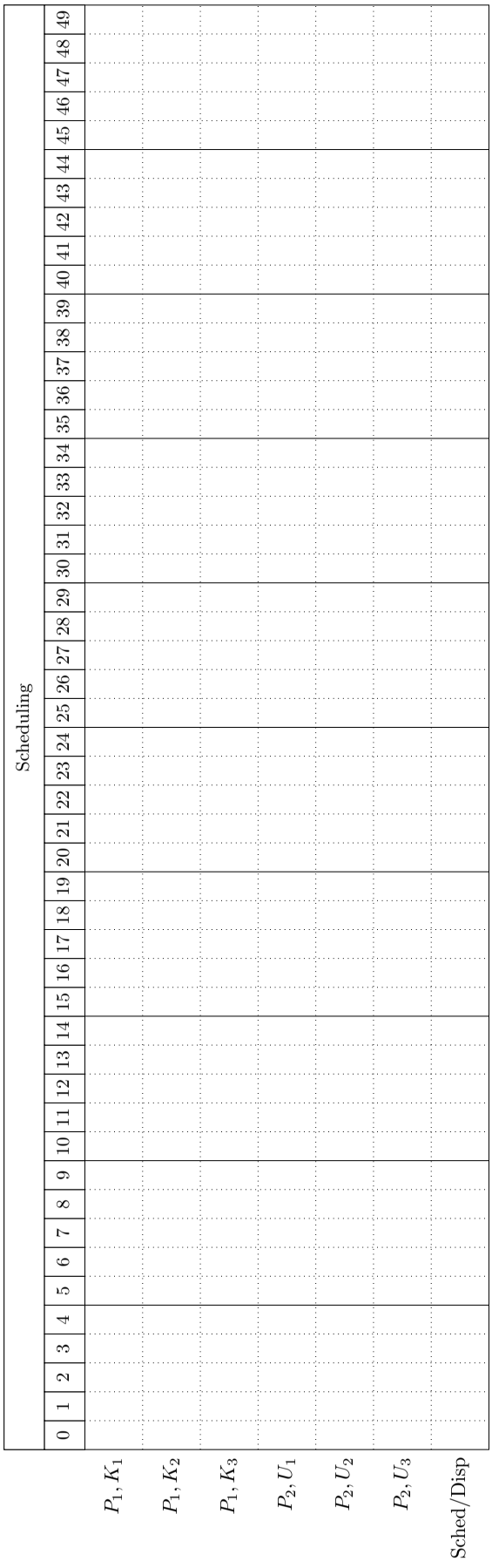
Der **User-Level-Scheduler** läuft **unabhängig** vom Kernel-Scheduler. Gehen Sie davon aus, dass jeder Thread nach **zwei Zeiteinheiten** oder beim Terminieren die Kontrolle an den User-Level-Scheduler abgibt.

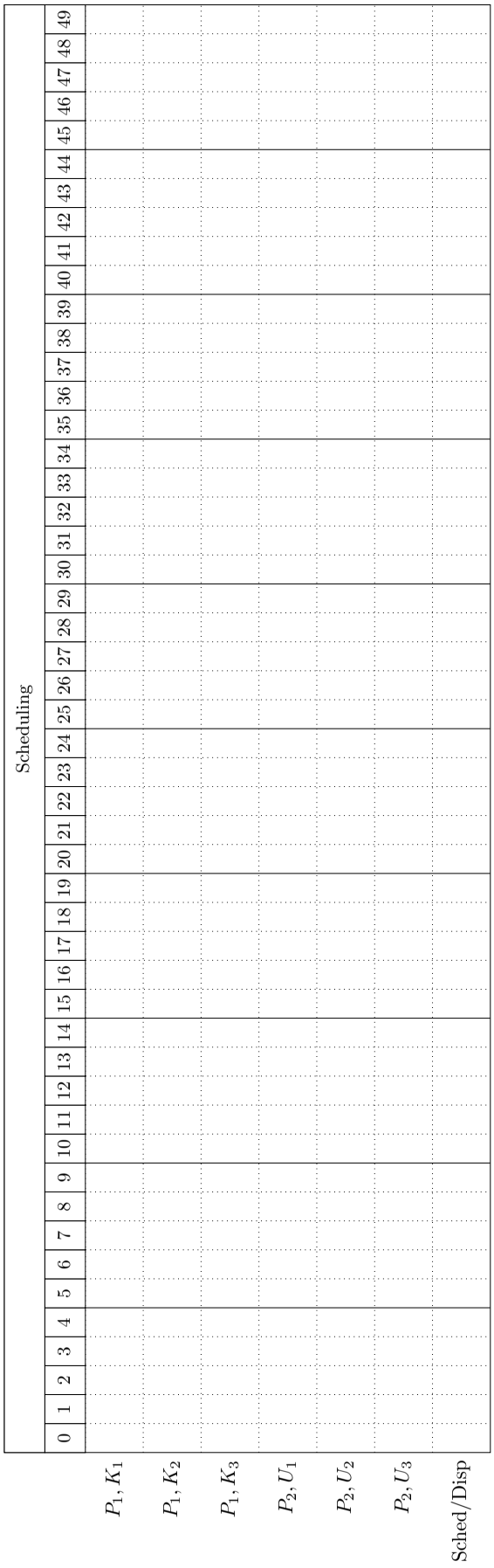
Der User-Level-Scheduler verwendet die **Round Robin Scheduling** Strategie. D.h. wie beim Kernel-Scheduler werden die Threads in der Reihenfolge ihrer Ankunft bearbeitet. Weiterhin startet der User-Level-Scheduler neue Threads sofort: Sobald ein neuer Thread hinzu kommt, gibt der laufende Thread die CPU ab und der neue Thread bekommt sie zugewiesen. Als nächstes ist dann der Thread an der Reihe, der regulär nach dem neuen Thread folgt.

Der Aufwand für das User-Level-Scheduling ist vernachlässigbar klein, kostet also in dieser Aufgabe **keine Zeiteinheiten**.

Skizzieren Sie unter diesen Annahmen den Ablauf der Prozesse/Threads im folgenden Gantt-Diagramm. Markieren Sie die Rechenzeiten mit einem **X** und die Wartezeiten mit einem **-**.

Noch Fragen? Dann schaut doch mal im Zulip-Stream vorbei:
`https://zulip.in.tum.de`





Noch Fragen? Dann schaut doch mal im Zulip-Stream vorbei:
`https://zulip.in.tum.de`

Aufgabe 2 Linux-Scheduling

Seit der Linux Kernel Version 2.6.23 wird der sogenannte **Completely-Fair-Scheduler** (CFS) verwendet, um zu entscheiden, welcher Prozess als nächstes an die CPU gebunden werden soll.[1] Im Folgenden werden wir uns mit diesem etwas vertrauter machen.

> The UNIX system has a command, *nice*, which allows a user to voluntarily reduce the priority of his process, in order to be nice to the other users. Nobody ever uses it.
>
> Andrew Tanenbaum

Der CFS unterstützt 40 statische Prioritätsstufen, mit denen festgelegt werden kann, welchen Anteil jeder Prozess an der vergeben CPU-Zeit bekommen soll.[2] Die Prioritäten des CFS werden indirekt durch den *niceness* Wert eines Prozesses bestimmt. Dieser Wert kann unter Linux (z.B. durch die Programme `top` oder `htop`) dynamisch an Prozesse vergeben werden und variiert zwischen -20 und $+19$, wobei kleinere Werte eine höhere Priorität, also einen geringeren Grad an „Nettigkeit“ bedeuten. Jedem niceness Wert ist ein Gewicht w zugeordnet, das im Linux Kernel (`kernel/sched/core.c`) im folgenden Array `sched_prio_to_weight` vorgegeben wird:

```
const int sched_prio_to_weight[40] = {
        /* -20 */     88761,     71755,     56483,     46273,     36291,
        /* -15 */     29154,     23254,     18705,     14949,     11916,
        /* -10 */      9548,      7620,      6100,      4904,      3906,
        /*  -5 */      3121,      2501,      1991,      1586,      1277,
        /*   0 */      1024,       820,       655,       526,       423,
        /*   5 */       335,       272,       215,       172,       137,
        /*  10 */       110,        87,        70,        56,        45,
        /*  15 */        36,        29,        23,        18,        15,
};
```

Dem Default niceness Wert 0 ist somit das Gewicht von 1024 zugeordnet.

Im Gegensatz zu Schedulingstrategien wie z.B. dem Round Robin werden keine festen Zeitquanten an die Prozesse vergeben. Stattdessen wird dynamisch (vor jeder Ausführung) ein Quantum bzw. eine **time slice** *TS* berechnet, der sowohl die Anzahl als auch die Prioritäten aller aktuell aktiver Prozesse auf dem System berücksichtigt. Bei der Berechnung der *time slice TS* wird der Wert der **targeted latency** *TL* mit in Betracht genommen. Dieser Wert stellt ein Intervall dar, in dem alle Prozesse mindestens einmal die CPU bekommen sollen.

Der *TL* Wert ist fest vorgegeben.[3] Die Berechnung der time slice ergibt sich bei *n aktiven* Prozessen durch die folgende Formel:

$$TS_i = TL \cdot \frac{w_i}{\sum_{j=1}^{n} w_j} \tag{1}$$

Zusätzlich zum vergebenen Zeitquantum muss der Scheduler natürlich auch entscheiden, welcher Prozess überhaupt als nächstes an die CPU gelassen werden soll. Normalerweise würde man hierfür vermutlich den Prozess auswählen, der bis dato am wenigsten Rechenzeit *rt* (**real runtime**) bekommen hat. Da wir hier allerdings berücksichtigen müssen, dass Prozesse mit höherer Priorität grundsätzlich länger an die CPU gelassen werden, verwenden wir stattdessen eine **virtual runtime** *vt*. Diese verrechnet die *real runtime* zusätzlich mit der Priorität des entsprechenden Prozesses.

Die *virtual runtime* vt_i für den *i*-ten Prozess P_i wird wie folgt berechnet:

$$vt_{i\_new} = vt_{i\_old} + \frac{1024}{w_i}(rt_{i\_new} - rt_{i\_old}) \tag{2}$$

Um den nächsten Prozess effizient finden zu können, verwaltet der Scheduler die Prozesse in einem Red-Black Tree[4] ansteigend nach ihrer *virtual runtime*.

[1] Mit Linux 6.6 (Oktober 2023) wurde der CFS nach gut 16 Jahren durch einen „Earliest Eligible Virtual Deadline First“ (EEVDF) Scheduler ersetzt. Dieser basiert auf dem CFS, ist aber wesentlich komplexer. Daher behandeln wir hier lediglich den CFS.

[2] In der Praxis hat Linux 140 Prioritäten. Die ersten 100 davon (0-99) sind für (weiche) Echtzeitprozesse reserviert, die oberen 40 (100-139) für normale Prozesse. Da ein regulärer Linux Kernel keine Unterstützung für Echtzeitsysteme bietet, vernachlässigen wir in dieser Aufgabe aber die unteren 100.

[3] Unter Linux lässt sich dieser Wert beim Booten einstellen. Per Default ist er 6ms $\cdot$ $(1 + \log_2(\texttt{NR\_CORES}))$.

[4] siehe `https://en.wikipedia.org/wiki/Red%E2%80%93black_tree`

Noch Fragen? Dann schaut doch mal im Zulip-Stream vorbei:
`https://zulip.in.tum.de`

In dieser Teilaufgabe sollen Sie sich das Linux-Scheduling an einem Beispiel verdeutlichen. Gegeben seien dazu folgende Prozesse mit ihrer Rechenzeit (in *ms*), dem niceness Wert und der entsprechenden Gewichtung:

Prozess	Rechenzeit	Niceness	Weight
1	20	0	1024
2	25	-5	3121
3	5	1	820
4	3	18	18
5	33	-10	9548

Zeichnen Sie die Abarbeitungsfolge der Prozesse, die sich bei Anwendung der oben erläuterten CFS Strategie ergibt. Verwenden Sie zur Hilfe eine Tabelle (siehe unten), die Spalten für die real runtime rt, time slice TS_i und virtual runtime vt für jeden der fünf Prozesse enthält. Die Zeilen der Tabelle ergeben den Zeitablauf des vorgestellten Scheduling Verfahrens.

Zur Vereinfachung nehmen Sie an, dass die **targeted latency** *TL* **50** *ms* beträgt.[5] Davon abgesehen werden sowohl die time slice TS_i als auch die virtual time vt_i auf ganze Zahlen **aufgerundet**. Beachten Sie, dass die time slice TS_i des i-ten Prozesses P_i das Zeit-Quantum seiner nächsten Ausführung vorgibt (z.B. wenn der Prozess P_i zum Zeitpunkt 0 mit dem time slice TS_i von 3 ausgeführt wird, so läuft P_i bis zum Zeitpunkt 3. Dieser Wert muss entsprechend zu rt_i addiert werden). In unserem vereinfachten Szenario beträgt der Vektor der **Ankunftszeiten am Scheduler** $\vec{a} = (0, 0, 0, 0, 0)$. Die Anfangswerte für rt_i und vt_i haben zum Zeitpunkt 0 den Wert 0.

Vervollständigen Sie die folgende Tabelle (die ersten 3 Zeilen sind bereits vollständig ausgefüllt).

t	rt_1	TS_1	vt_1	rt_2	TS_2	vt_2	rt_3	TS_3	vt_3	rt_4	TS_4	vt_4	rt_5	TS_5	vt_5
0	0	4^{*_1}	0	0		0	0		0	0		0	0		0
4	4	4	4	0	11	0	0		0	0		0	0		0
15	4		4	11	11	4^{*_2}	0		0	0		0	0		0

Beispiele:
$*_1$: $TS_1 = 50\,\text{ms} \cdot \frac{1024}{1024+3121+820+18+9548} \approx 3{,}52\,\text{ms} \rightarrow 4$
$*_2$: $vt_2 = 0\,\text{ms} + \frac{1024}{3121} \cdot (11\,\text{ms} - 0\,\text{ms}) \approx 3{,}61\,\text{ms} \rightarrow 4$

Wenn Sie sich über den Rahmen der Tutorübung hinaus noch weiter mit der Funktionsweise des Linux-Schedulers auseinandersetzen wollen, empfehlen wir Ihnen die entsprechende man-Page (`man sched`*) sowie die Dokumentation im Source-Code des Linux-Kernels (*`Documentation/scheduler/sched-design-CFS.txt`*). (Die eigentliche Implementierung des CFS-Schedulers finden Sie in der Datei* `kernel/sched/fair.c`*.)*

[5] Der Linux-Scheduler rechnet hierbei übrigens in Nanosekunden.

Noch Fragen? Dann schaut doch mal im Zulip-Stream vorbei:
`https://zulip.in.tum.de`

Aufgabe 3 Zusätzliche Übungsaufgaben (Optional)

Falls Sie die vorgestellten Scheduling-Verfahren noch mehr üben möchten, können Sie hierfür unser Online-Übungstool verwenden:

`http://scheduling.tum.sexy`

Desweiteren finden Sie hier noch einige Aufgaben zum Thema **Prozessverwaltung** und **Scheduling** aus den Klausuren der vergangenen Semester:

- Midterm 2020: Aufgabe 3
- Midterm 2019: Aufgabe 2
- Endterm 2020: Aufgabe 7
- Endterm 2019: Aufgabe 4
- Endterm 2017: Aufgabe 4
- Endterm 2016: Aufgabe 3
- Retake 2019: Aufgabe 7
- Retake 2017: Aufgabe 4
- Retake 2016: Aufgabe 2

Noch Fragen? Dann schaut doch mal im Zulip-Stream vorbei:
`https://zulip.in.tum.de`