Lehrstuhl Connected Mobility
Fakultät für Informatik
Technische Universität München

Grundlagen von Betriebssystemen und Systemsoftware

WS 2018/19

Übung 4: Prozesse und Threads

zum 12. November 2018

- Die **Hausaufgaben** für dieses Übungsblatt müssen **spätestens** am **Sonntag, den 18.11.2018 um 23:59** in Moodle hochgeladen worden sein.
- Alle C-Programme müssen mit den folgenden Flags kompiliert werden:
  `gcc -Wall -o <progname> <prog.c>`.
- Hierzu stellen wir für jedes Übungsblatt jeweils ein Makefile bereit, das **nicht** verändert werden darf, um sicherzustellen, dass Ihre Abgabe auch korrekt von uns getestet und bewertet werden kann. **Ihr Programmcode muss zwingend mit dem Makefile kompilierbar sein. Anderenfalls kann die Abgabe nicht bewertet werden.** Wenn Sie zwischenzeitlich Änderungen vornehmen wollen, um etwa bestimmte Teile mittels `#ifdef` einzubinden und zu testen, dann kopieren Sie am besten das Makefile und modifizieren Ihre Kopie. Mit `make -f <Makefile>` können Sie dann eine andere Datei zum Übersetzen verwenden.
- Die Abgabe der Programme erfolgt als Archivdatei, die die verschiedenen Quelldateien (.{c|h}) umfasst und **nicht** in Binärform. Nicht kompilierfähiger Quellcode wird **nicht gewertet**.
- Um die Abgabe zu standardisieren enthält das Makefile ein Target "submit" (`make submit`), was dann eine Datei blatt04.tar zum Hochladen erzeugt.
- Damit die richtigen Dateien hochgeladen und ausgeführt werden, geben wir bei allen Übungen die jeweils zu verwendenden Dateinamen für den Quellcode und auch für das Executable an.
- Die Tests werden auf diesem Aufgabenblatt mittels Python-Programmen durchgeführt. Zum Ausführen der Tests geben Sie `python xyz_test.pyc` auf der Konsole ein. Ihre ausführbaren Programme müssen sich im selben Verzeichnis wie die Testprogramme befinden.

1 Vorbereitung auf das Tutorium

Was ist die Rolle der folgenden POSIX- (Portable Operating System Interface) bzw. Bibliotheksfunktionen? Erläutern Sie kurz die Parameter und den Rückgabewert jeder Funktion. [1]

- `int pthread_create (pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);`
- `void pthread_exit (void *value_ptr);`
- `int pthread_join (pthread_t thread, void **value_ptr);`
- `int phtread_cancel (pthread_t thread);`
- `pthread_t pthread_self ();`
- `int fprintf(FILE *restrict stream, const char *restrict format, ...);`
- `int fscanf(FILE *restrict stream, const char *restrict format, ...);`

2 Tutoraufgaben

2.1 Scheduler & Dispatcher

Um die Ausführung von mehreren Prozessen auf einem Prozessor zu realisieren, müssen strategische Entscheidungen getroffen werden, um einen bestimmten *rechenwilligen* Prozess (oder Thread) für eine bestimmte Zeit an die CPU zu *binden* bzw. einen *rechnenden* Prozess von der CPU zu *entbinden* bzw. zu *lösen*. Die Entscheidung welcher Prozess als nächstes ausgeführt werden soll wird vom Scheduler getroffen. Der eigentliche Kontextwechsel zwischen dem *rechnenden* und dem vom Scheduler ausgewählten *rechenwilligen* Prozess wird durch den Dispatcher realisiert. Um den Zusammenhang zwischen dem Scheduler und dem Dispatcher besser zu verstehen, beschäftigen wir uns in dieser Aufgabe zunächst mit allgemeinen Fragen zum Scheduler und dem Dispatcher. Anschließend betrachten wir einige ausgewählte Schedulingstrategien und versuchen anhand von Kenngrößen ihre Vor- und Nachteile nachzuvollziehen.

2.1.1 Allgemeine Fragen zu Scheduling

Beantworten Sie die folgenden Fragen. Kurze, stichpunktartige Antworten sind hierfür ausreichend.

a) Was ist der Unterschied zwischen **preemptive** (unterbrechenden) und **non-preemptive** (nichtunterbrechenden) Schedulingstrategien?

b) Warum ist es wichtig dass der Scheduler zwischen I/O- und CPU-bound Prozessen unterscheidet?

c) Bei Welchem der folgenden Scheduling Algorithmen können Prozesse verhungern? Kann man dieses Problem umgehen? Falls ja, wie?

- First-Come-First-Served
- Shortest Job First
- Round Robin

[1] Erläuterungen und Beispiele der einzelnen Funktionen finden Sie auch in den **man**-Pages auf Ihrer Linux-VM.

- Priority Scheduling (mit statischen Prioritäten)

d) Können Sie sich ein Szenario vorstellen, in dem ein und der selbe Prozess mehrmals in die Run-Queue (bzw. Ready-Queue) des Round Robin basierten Schedulers eingefügt wird?

2.1.2 Allgemeine Fragen zum Dispatcher

Machen Sie sich mit dem Zusammenhang von Scheduler und Dispatcher vertraut und beantworten Sie folgende Fragen. Kurze, stichpunktartige Antworten sind hierfür ausreichend.

a) Was ist ein Prozesskontrollblock (PCB)? Was ist eine Threadtabelle (Thread-Control-Block)?

b) Welche weiteren Datenstrukturen (außer dem PCB) werden für einen Prozesswechsel benötigt?

c) Worin liegt der Unterschied zwischen dem Dispatchen von Prozessen und dem Dispatchen von Threads? Gehen Sie dabei insbesondere auf User-Level und Kernel-Level Threads ein.

d) Grenzen Sie die Aufgabe des Dispatchers von der des Schedulers ab. Wie gestaltet sich die Zusammenarbeit dieser beiden Komponenten? Welche Bedeutung hat der PCB für das Scheduling? Welche für das Dispatching?

e) Beschreiben Sie den Ablauf eines Kontextwechsels bzw. des Dispatching eines Prozesses.

f) Welche Rolle spielt die **Größe** des Prozesskontrollblocks beim Kontextwechsel? Betrachten Sie diesem Aspekt im Hinblick auf **Häufigkei**t und **Effizienz** multipler Kontextwechsel.

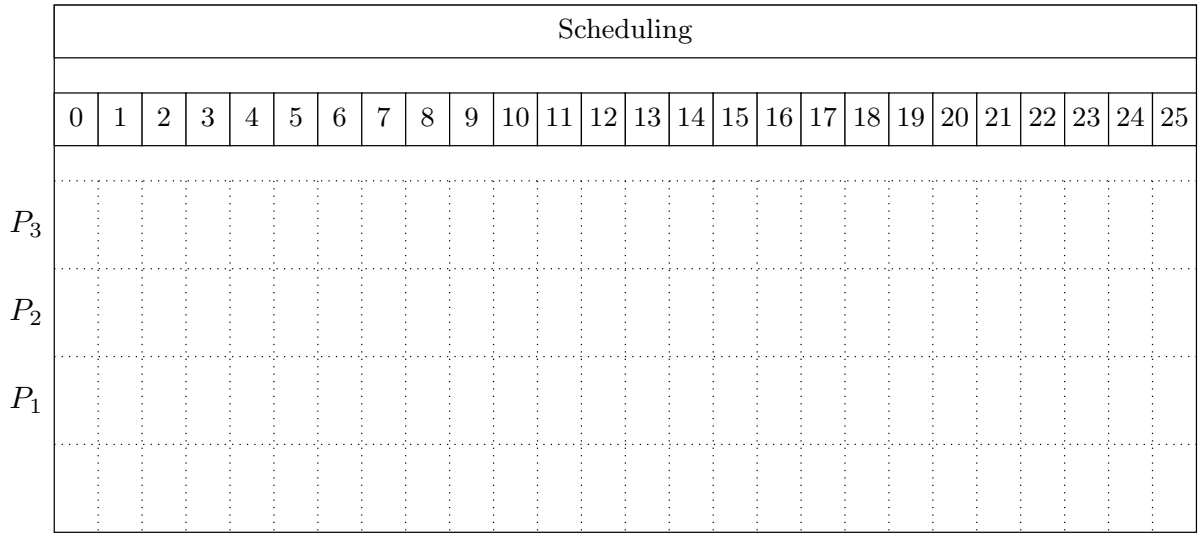
2.2 Schedulingstrategien

Es seien 3 Prozesse gegeben. Der Vektor ihrer Ankunftszeiten am Scheduler beträgt $\vec{a} = (0, 5, 2)$. Ihre Rechenzeiten sind durch $\vec{r} = (7, 3, 4)$ gegeben. Nehmen Sie an, dass ein Kontextwechsel eine Zeiteinheit benötigt. Modellieren Sie den Scheduler/Dispatcher als einen eigenständigen Prozess.

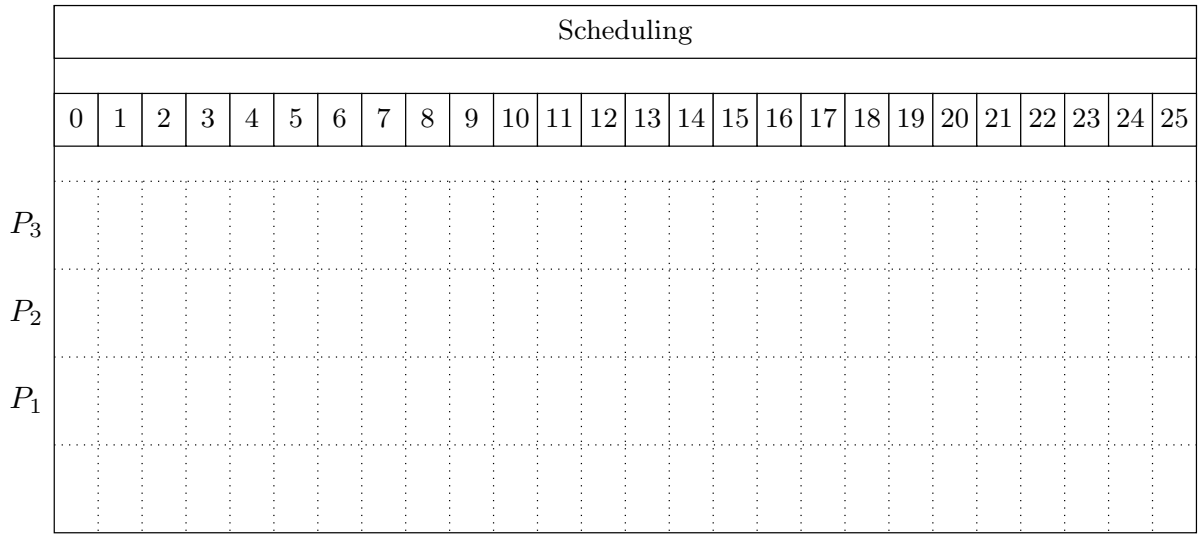
Skizzieren Sie unter diesen Annahmen den Ablauf der Prozesse in einem Gantt-Diagram für folgende Schedulingstrategien.

Gantt-Diagramm: Verwenden Sie die x-Achse des Diagramms für die zeitliche Dimension und die y-Achse für die Prozesse. Stellen Sie für jeden Prozess die Wartezeit in Form einer gestrichelten Linie (beginnend mit der Ankunftszeit des Prozesses), und die Rechenzeit in Form einer ununterbrochenen Linie (bzw. eines Balkens) dar. Berücksichtigen Sie die Prioriäten in Ihrem Diagramm.

a) First-Come-First-Served (FCFS): Non-preemptive, Prozesse werden in der Reihenfolge ihrer Ankunftszeiten abgearbeitet.

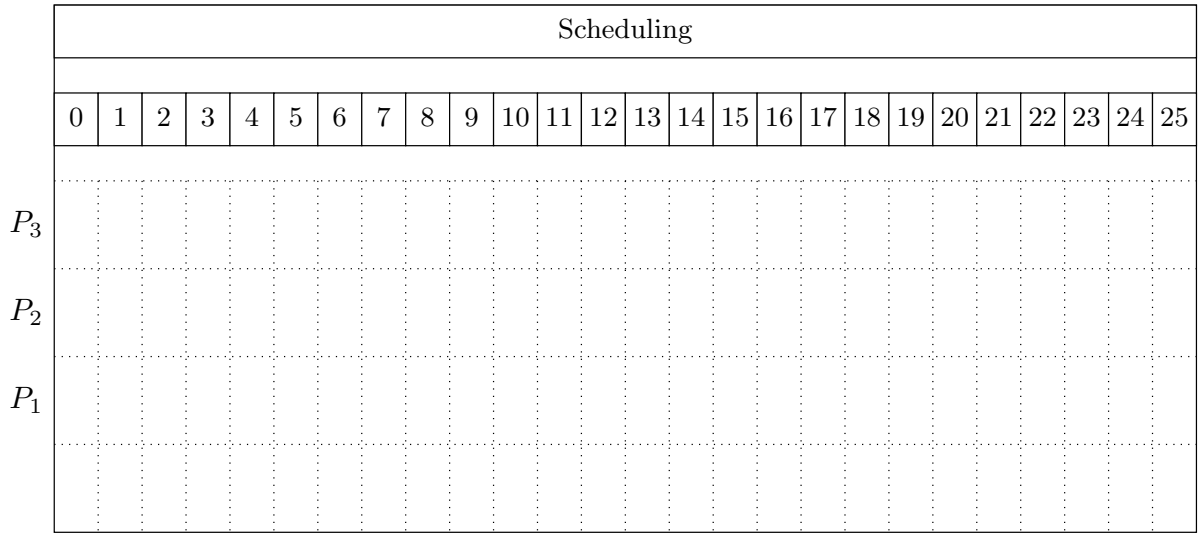


b) Shortest Remaining Time Next (SRTN): Preemptive, Auswahl des Prozesses mit der kürzesten verbleibenden Rechenzeit, Unterbrechungen erfolgen nur beim Eintreffen eines neuen Prozesses.

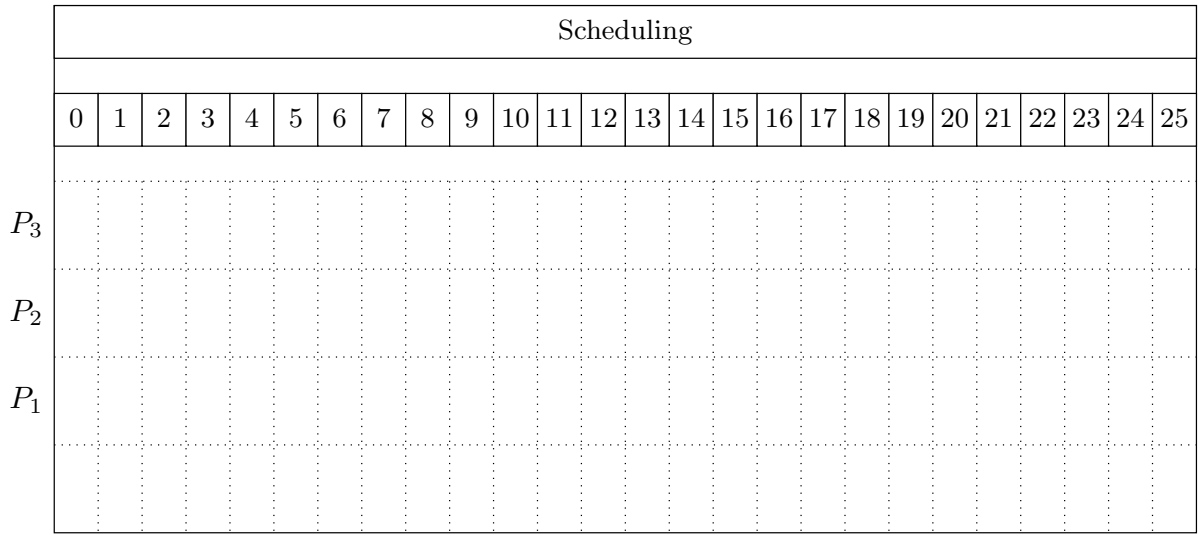


c) Round-Robin mit einem Zeitquantum von einer Zeiteinheit und zyklischer[2] Abarbeitung der Prozesse (gleiche und nicht veränderbare Prioritäten)

[2] Sortierung nach PID der Prozesse



d) Round-Robin mit einem Zeitquantum von 2 Zeitenheiten und zyklischer Abarbeitung der Prozesse (gleiche und nicht veränderbare Prioritäten)



3 Hausaufgaben

3.1 Erzeugen mehrerer Threads: threadit (3 Punkte)

Verändern Sie Ihr Programm `forkmany.c` von Übung 3, so dass es Threads statt Prozesse erzeugt, sonst aber die gleiche Funktionalität bietet. Statt der Prozess-ID soll dann die Thread-ID ausgegeben werden.

Auch hier sollen N Threads erzeugt, die alle jeweils bis K zählen. Der Main-Thread soll sich alle erzeugten Threads in einer Liste merken und warten, bis sie alle beendet sind.

Auch hier sollen die Threads nicht alle bis K zählen, sondern bis zu einer zufälligen ganzen Zahl in $[K/2; 1,5 \times K]$, wenn der optionale Parameter "-r" auf der Kommandozeile angegeben wird. Hierfür können Sie Integer-Arithmetik verwenden. Nehmen Sie als Default-Parameter $K = 10$ und $N = 1$ an, falls der jeweilige Parameter nicht auf der Kommandozeile übergeben wird.

Der Exit-Code muss hier nicht mehr ausgegeben werden. Ansonsten bleibt das Ausgabeformat unverändert wie in Übung 3. Führen Sie das Programm mehrfach mit vielen Threads ($K \geq 10$) aus und beobachten Sie, ob/wie sich die Reihenfolge der Ausgaben der Threads bei den einzelnen Schritten aufgrund des Schedulings ändert.

Hinweis: die eckigen Klammern in der folgenden Syntax drücken optionale Parameter aus. Die Reihenfolge der Argumente kann prinzipiell beliebig sein. Verwenden Sie `getopt (3)` zum Parsieren der Kommandozeilenparameter.

Verwenden Sie die folgende Funktion für die Ausgabe:

```
printf ("%10u\t%d\t%ld\n", (unsigned int) pthread_self (), getpid (), i);
```

Syntax: `threadit [-k <K>] [-n <N>] [-r]`

Quellen: `threadit.c list.c list.h`

Executable: `threadit`

Beispielausgabe:

```
$ ./threadit -k 6 -n 3 -r
Start: Wed Nov  7 06:02:43 2018
1571497728      1067    1
1579890432      1067    1
1588283136      1067    1
1571497728      1067    2
1579890432      1067    2
1588283136      1067    2
1571497728      1067    3
1579890432      1067    3
1588283136      1067    3
1571497728      1067    4
1579890432      1067    4
1588283136      1067    4
1571497728      1067    5
1571497728      1067    6
1571497728      1067    7
Ende: Wed Nov  7 06:02:50 2018
```

3.2 Simulation von Thread-Scheduling: threadsched (7 Punkte)

Ergänzen Sie Ihre Datenstruktur, die die Threads verwaltet, um die folgenden Daten:

- laufende Nummer des Threads (1, .., N)
- Thread-Priorität
- Startzeitpunkt des Threads
- Zeit, die der Thread bereits gelaufen ist
- Ziellaufzeit des Threads (Approximieren Sie die Ziellaufzeit mit Hilfe der Zahl, bis zu der der Thread zählen soll.)

Überarbeiten Sie Ihr Programm aus Abschnitt 3.1 so, dass es

- keine echten Threads mehr erzeugt, sondern nur die Einträge in der Liste erzeugt. Starten Sie keine echten Threads, sondern nutzen Sie das Programm nur zur Simulation des Ablaufs. Das heißt, implementieren Sie einen einfachen preemptive Scheduler, der den Threads entsprechend ihrer Startzeiten, Prioritäten und Laufzeiten die CPU zuweist. Gehen Sie davon aus, dass nur eine CPU vorhanden ist, also immer nur ein Thread zur Zeit aktiv sein kann.
- die entsprechenden Parameter von der Standardeingabe (stdin) liest. Lesen Sie die Parameter Zeile für Zeile ein, bis N vollständige Einträge gelesen wurden. Führen Sie sinnvolle Parameterbereiche ein: $N \leq 10$, Startzeit ≤ 100000 (100s), Laufzeit ≤ 30000 (30s).

Dazu soll folgende Syntax verwendet werden:

Syntax: `threadsched -n <N Threads> -t <time step> -q <time quantum> -a <algorithm>`

Das Zeitquantum definiert die Zeitschritte in Ihrer Simulation. Zählen Sie in Ihrer Simulation – beginnend mit 0 – eine Uhr in diesen Zeitquanten hoch. Dies wird für die Ausgabe verwendet, s. Gantt-Chart unten.

Das Format für die Eingabe auf STDIN ist dann wie folgt:

```
Prio Startzeit Laufzeit-fuer-Thread-1<newline>
Prio Startzeit Laufzeit-fuer-Thread-2<newline>
...
Prio Startzeit Laufzeit-fuer-Thread-N<newline>
```

Die einzelnen Eingabefelder sind wie folgt definiert:

Prio: Die Priorität prio ist ein Integer aus dem Intervall [1 ; 10]. 1 ist die höchste Priorität, 10 die niedrigste.

Startzeit: Die Startzeit gibt die Zeitspanne an (in Zeiteinheiten, z.B. Millisekunden), die der Thread nach Aufruf des Programms gestartet wird. Die Startzeiten müssen nicht monoton steigend angegeben werden. Sollten zwei oder mehr Startzeiten gleich sein, starten Sie die Threads unmittelbar hintereinander zum gleichen Zeitpunkt in der Reihenfolge, in der sie angegeben worden sind.

Laufzeit: Die Laufzeit gibt die Dauer an, die ein Thread laufen wird. Dabei wird als Laufzeit nur die Zeit gewertet, die der Prozess tatsächlich die CPU zugewiesen bekommen hat.

Implementieren Sie die folgenden drei Scheduling-Algorithmen:

a) **Round-robin** (“RR”): dieser Algorithmus ignoriert die Prioritäten und führt die zu einem Zeitpunkt verfügbaren Threads immer einen nach dem anderen aus, jeweils für das entsprechende Zeitquantum. Wenn ein neuer Thread gestartet wird, wird dieser hinten an die aktuelle Liste rechenbereiter Threads angehängt.

b) **Priority Round-robin** (“PRR”): Dieser Algorithmus soll die Prioritäten berücksichtigen und strikt die Threads mit höherer Priorität vor denen mit niedriger Priorität ausführen. Innerhalb einer Priorität wird Round-Robin-Scheduling verwendet.

c) **Shortest Remaining Time Next** (“SRTN”): Dieser Algorithmus soll die Threads entsprechend ihrer verbleibenden Laufzeit priorisieren, wobei derjenige mit der kürzesten Restlaufzeit zuerst die CPU erhält. Das System soll preemptive arbeiten: wenn ein neuer Thread hinzukommt, der eine **kürzere** Restlaufzeit aufweist, wird der gerade laufende unterbrochen und der neue ausgeführt; d.h. bei gleicher Restlaufzeit wird läuft der aktive Thread weiter.

Visualisieren Sie den Ablauf durch ein Gantt-Chart in ASCII-Art. Schreiben Sie eine Funktion, die die aktuelle (relative Zeit) anzeigt und, die laufende Nummer des simulierten Threads in der richtigen Spalte ausgibt. Sehen Sie immer 10 Threads vor (s. erste Zeile des Gantt-Charts), gleichgültig wie viele Threads simuliert laufen. Verwenden Sie jeweils zwei Leerzeichen zwischen den Thread-Nummern. Die Ausgabe soll genau wie folgt aussehen.

```
  Time | 1  2  3  4  5  6  7  8  9  10
-------+-------------------------------
000000 |
000100 | 1
000200 | 1
000300 | 1
000400 |          4
000500 |          4
000600 |          4
```

In diesem Beispiel läuft Thread 1 von 100 bis 300ms und dann Thread 4 von 400 bis 600ms.

Verwenden Sie folgende Funktion für die Ausgabe, damit sichergestellt ist, dass Ihre Ausgabe den Erwartungen des Testprogramms entspricht und es keine fehlerhafte Bewertung gibt.

```
void print_time_step (int time, int thread_num) {
    static int    first_time = 1;
    int           i;

    if (first_time) {
        printf ("  Time |  1  2  3  4  5  6  7  8  9  10\n");
        printf ("-------+-------------------------------\n");
        first_time = 0;
    }
    printf ("%06d |", time);
    if (thread_num) {
        for (i = 1; i < thread_num; i++)
            printf ("   ");
        printf ("  %d\n", thread_num);
    } else
        printf ("\n");
}
```

Führen Sie immer **erst** das Scheduling aus mit evtl. Aktualisierung des aktuell ausgeführten Threads, und rufen Sie **dann** die Ausgabefunktion auf. Das Programm soll terminieren, sobald der letzte

Thread beendet ist. Denken Sie daran, dass es durchaus sein kann, dass zwischenzeitlich kein Thread aktiv ist, aber noch weitere in Zukunft zur Ausführung kommen werden.

Quellen: `threadsched.c list.c list.h`

Executable: `threadsched`

Beispielausgaben zum Testen:

```
$ ./threadsched -n 3 -t 10 -q 50 -a RR
1 100 200
1 200 200
1 300 200
  Time |  1  2  3  4  5  6  7  8  9  10
-------+-------------------------------
000000 |
000010 |
000020 |
000030 |
000040 |
000050 |
000060 |
000070 |
000080 |
000090 |
000100 |  1
000110 |  1
000120 |  1
000130 |  1
000140 |  1
000150 |  1
000160 |  1
000170 |  1
000180 |  1
000190 |  1
000200 |     2
000210 |     2
000220 |     2
000230 |     2
000240 |     2
000250 |  1
000260 |  1
000270 |  1
000280 |  1
000290 |  1
000300 |     2
000310 |     2
000320 |     2
000330 |     2
000340 |     2
000350 |        3
000360 |        3
000370 |        3
000380 |        3
000390 |        3
000400 |  1
000410 |  1
000420 |  1
000430 |  1
```

```
000440 |  1
000450 |     2
000460 |     2
000470 |     2
000480 |     2
000490 |     2
000500 |        3
000510 |        3
000520 |        3
000530 |        3
000540 |        3
000550 |     2
000560 |     2
000570 |     2
000580 |     2
000590 |     2
000600 |        3
000610 |        3
000620 |        3
000630 |        3
000640 |        3
000650 |        3
000660 |        3
000670 |        3
000680 |        3
000690 |        3
```

```
$ ./threadsched -n 4 -t 10 -q 50 -a PRR
1 50 200
1 100 200
2 150 200
1 200 200
  Time |  1  2  3  4  5  6  7  8  9  10
-------+-------------------------------
000000 |
000010 |
000020 |
000030 |
000040 |
000050 |  1
000060 |  1
000070 |  1
000080 |  1
000090 |  1
000100 |     2
000110 |     2
000120 |     2
000130 |     2
000140 |     2
000150 |  1
000160 |  1
000170 |  1
000180 |  1
000190 |  1
000200 |     2
000210 |     2
000220 |     2
000230 |     2
```

```
000240 |       2
000250 |               4
000260 |               4
000270 |               4
000280 |               4
000290 |               4
000300 |   1
000310 |   1
000320 |   1
000330 |   1
000340 |   1
000350 |       2
000360 |       2
000370 |       2
000380 |       2
000390 |       2
000400 |               4
000410 |               4
000420 |               4
000430 |               4
000440 |               4
000450 |   1
000460 |   1
000470 |   1
000480 |   1
000490 |   1
000500 |       2
000510 |       2
000520 |       2
000530 |       2
000540 |       2
000550 |               4
000560 |               4
000570 |               4
000580 |               4
000590 |               4
000600 |               4
000610 |               4
000620 |               4
000630 |               4
000640 |               4
000650 |           3
000660 |           3
000670 |           3
000680 |           3
000690 |           3
000700 |           3
000710 |           3
000720 |           3
000730 |           3
000740 |           3
000750 |           3
000760 |           3
000770 |           3
000780 |           3
000790 |           3
000800 |           3
000810 |           3
```

```
000820 |          3
000830 |          3
000840 |          3
```

```
$ ./threadsched -n 3 -t 10 -q 50 -a SRTN
1 100 300
1 150 200
1 200 100
  Time |  1  2  3  4  5  6  7  8  9  10
-------+-------------------------------
000000 |
000010 |
000020 |
000030 |
000040 |
000050 |
000060 |
000070 |
000080 |
000090 |
000100 |  1
000110 |  1
000120 |  1
000130 |  1
000140 |  1
000150 |     2
000160 |     2
000170 |     2
000180 |     2
000190 |     2
000200 |        3
000210 |        3
000220 |        3
000230 |        3
000240 |        3
000250 |        3
000260 |        3
000270 |        3
000280 |        3
000290 |        3
000300 |     2
000310 |     2
000320 |     2
000330 |     2
000340 |     2
000350 |     2
000360 |     2
000370 |     2
000380 |     2
000390 |     2
000400 |     2
000410 |     2
000420 |     2
000430 |     2
000440 |     2
000450 |  1
000460 |  1
000470 |  1
```

```
000480 |   1
000490 |   1
000500 |   1
000510 |   1
000520 |   1
000530 |   1
000540 |   1
000550 |   1
000560 |   1
000570 |   1
000580 |   1
000590 |   1
000600 |   1
000610 |   1
000620 |   1
000630 |   1
000640 |   1
000650 |   1
000660 |   1
000670 |   1
000680 |   1
000690 |   1
```