– Lösung zur Praktikumsaufgabe 7 –

Thema: *Ratenmonotones Scheduling*

1.

$$u = \frac{1}{8} + \frac{3}{15} + \frac{4}{20} + \frac{6}{22} = \frac{351}{440} \approx 0.80$$

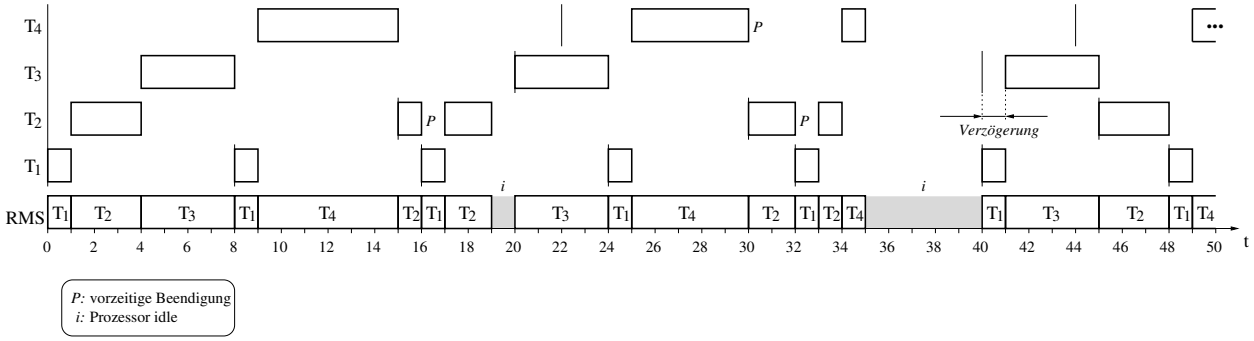

Abbildung 1: Ratenmonotoner Schedule für $T_1(8,1)$, $T_2(15,3)$, $T_3(20,4)$ und $T_4(22,6)$

2. a)

- Aufstellen des Plans für H ist aufwändig, da $H = \text{kgV}(10, 15, 25) = 150$.
- keine einfach periodische Taskmenge
- Liu/Layland:

$$u = \frac{2}{10} + \frac{5}{15} + \frac{9}{25} = \frac{67}{75} \approx 0.893 \not\leq u_{RM}(3) = 3\left(\sqrt[3]{2} - 1\right) = 0.779$$

- Es ist eine ratenmonotone Analyse nötig. Wir bestimmen die maximale Antwortzeit $t_{\text{maxresp},3}$ für $T_3(25, 9)$. Ist diese kleiner als die Deadline von T_3, dann ist die *gesamte* Taskmenge planbar.

$$\begin{aligned}
t^{(l+1)} &= 9 + \left\lceil \frac{t^{(l)}}{10} \right\rceil \cdot 2 + \left\lceil \frac{t^{(l)}}{15} \right\rceil \cdot 5 \\
t^{(0)} &= 9 \\
t^{(1)} &= 9 + \left\lceil \frac{9}{10} \right\rceil \cdot 2 + \left\lceil \frac{9}{15} \right\rceil \cdot 5 = \underline{16} \\
t^{(2)} &= 9 + \left\lceil \frac{16}{10} \right\rceil \cdot 2 + \left\lceil \frac{16}{15} \right\rceil \cdot 5 = \underline{23} \\
t^{(3)} &= 9 + \left\lceil \frac{23}{10} \right\rceil \cdot 2 + \left\lceil \frac{23}{15} \right\rceil \cdot 5 = \underline{25} \\
t^{(4)} &= 9 + \left\lceil \frac{25}{10} \right\rceil \cdot 2 + \left\lceil \frac{25}{15} \right\rceil \cdot 5 = \underline{25} = t_{\text{maxresp},3}
\end{aligned}$$

Die maximale Antwortzeit von (25,9) beträgt demnach 25 Zeiteinheiten; damit ist die Taskmenge (gerade noch so) ratenmonoton einplanbar. Jede noch

so kleine Verlängerung einer Ausführungszeit macht die Taskmenge unplanbar nach RMS.

b)
- $u = \frac{2}{10} + \frac{5}{12} + \frac{4}{15} = \frac{53}{60} \approx 0.883$ (kleiner als in Aufgabe 2.a)!)
- analog wie 2.a); Iteration:

$$\begin{aligned} t^{(l+1)} &= 4 + \left\lceil \frac{t^{(l)}}{10} \right\rceil \cdot 2 + \left\lceil \frac{t^{(l)}}{12} \right\rceil \cdot 5 \\ t^{(0)} &= 4 \\ t^{(1)} &= 11 \\ t^{(2)} &= 13 \\ t^{(3)} &= 18 \\ t^{(4)} &= \underline{18} = t_{\text{maxresp},3}. \end{aligned}$$

Die maximale Antwortzeit von (15,4) überschreitet die Deadline von 15 $\rightsquigarrow$ die Taskmenge ist nicht ratenmonoton einplanbar.

3. a) Eine Lösung zeigt das zugehörige Listing 1.

Listing 1: Werkzeug zur Ratenmonotonen Analyse

```
/*
    tda.c
    o performs a rate-monotonic analysis on a set of tasks
    o task descriptions are read from stdin and have the form

      T1(0, 5, 1, 5)
      T2(0, 7, 1, 9)
      etc.

    - tasks are ordered by increasing periods (decreasing priority
       )
    - tasks have a unique number following the 'T'
    - the four parameters are t_phi, t_p, t_e, t_d
    - NO SYNTAX CHECKING IS DONE. THE PROGRAM TRUSTS YOU!
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#define MAXTASKS 5000
#define MAXLINE  1024

struct periodic_task {
  double t_phi;  /* phase of task */
  double t_p;    /* period */
  double t_e;    /* execution time */
  double t_d;    /* deadline */
  double u_i;    /* load */
  double t_mr;   /* maximum response time */
```

```
  char *ok;       /* "pass" or "fail" */
};

struct periodic_task tasks[MAXTASKS];
int numTasks=0;

void read_taskset(void)
{
  char linebuf[MAXLINE], *aktpos;
  long index;

  while(!feof(stdin)) {
    if (!fgets(linebuf, MAXLINE-1, stdin)) {
      continue;
    }
    aktpos = strchr(linebuf, 'T');
    if (aktpos==NULL) {
      continue;         /* skip lines without 'T' */
    }

    aktpos = strtok(aktpos+1, "(");         /* find start of index
        */
    index = strtol(aktpos, NULL, 10) - 1;
    aktpos = strtok(NULL, ",");         /* find 1st parameter */
    tasks[index].t_phi = strtod(aktpos, NULL);
    aktpos = strtok(NULL, ",");         /* find 2nd parameter ...
        */
    tasks[index].t_p = strtod(aktpos, NULL);
    aktpos = strtok(NULL, ",");
    tasks[index].t_e = strtod(aktpos, NULL);
    aktpos = strtok(NULL, ",");
    tasks[index].t_d = strtod(aktpos, NULL);
    tasks[index].u_i = tasks[index].t_e / tasks[index].t_p;
    numTasks++;
    if (numTasks >= MAXTASKS) {
      printf("Max number of tasks (%d) exceeded. Please re-compile
          .\n", MAXTASKS);
      exit(EXIT_FAILURE);
    }
  }
  return;
}

void print_taskset(void)
{
  int c;
  double u=0.0;

  printf(" Task     t_phi     t_p      t_e     t_d      u_i
      t_mr    ok\n");
  for (c=0; c<numTasks; c++) {
    u += tasks[c].u_i;
    printf("T(%03d)   %5.2f    %5.2f    %5.2f    %5.2f   %5.4f
        %5.2f    %s\n",
     c+1, tasks[c].t_phi, tasks[c].t_p, tasks[c].t_e, tasks[c].t_d
```

```
          ,
       tasks[c].u_i, tasks[c].t_mr, tasks[c].ok);
    }
    printf("u = %.4f\n", u);
}

void calculate_maxresp(int idx)
{
    int higher;
    double itold, it;   /* past and current iteration value */

    itold=0;
    it = tasks[idx].t_e;
    while (itold != it) {
      itold = it;
      it = tasks[idx].t_e;
      for (higher=0; higher<idx; higher++) {
        it += ceil(itold / tasks[higher].t_p) * tasks[higher].t_e;
      }
    }
    tasks[idx].t_mr = it;

}

void test_feasibility(void)
{
    int idx;

    for (idx=0; idx<numTasks; idx++) {
      if (tasks[idx].t_mr <= tasks[idx].t_d) {
        tasks[idx].ok = "pass";
      }
      else {
        tasks[idx].ok = "fail";
      }
    }
}


int main(int argc, char* argv[])
{
    int index;

    read_taskset();

    for (index=0; index<numTasks; index++) {
      calculate_maxresp(index);
    }

    test_feasibility();
    print_taskset();

    exit(EXIT_SUCCESS);
}
```

Die Ausgabe für die gegebene Taskmenge ist:

```
robge@ipo51:~/txt/job/htw/ezs/src/prak07$ cat tasks6b.txt
T1(0, 5, 1, 5)
T2(0, 7, 1, 9)
T3(0, 10, 3, 10)
T4(0, 35, 7, 35)
T5(0, 71, 11, 71)
robge@ipo51:~/txt/job/htw/ezs/src/prak07$ ./tda < tasks6b.txt
 Task     t_phi     t_p      t_e     t_d      u_i     t_mr    ok
T(001)    0.00     5.00     1.00     5.00   0.2000    1.00    pass
T(002)    0.00     7.00     1.00     9.00   0.1429    2.00    pass
T(003)    0.00    10.00     3.00    10.00   0.3000    5.00    pass
T(004)    0.00    35.00     7.00    35.00   0.2000   20.00    pass
T(005)    0.00    71.00    11.00    71.00   0.1549   70.00    pass
u = 0.9978
```

Obwohl die Last sehr hoch ist, kann die Taskmenge nach RMS geplant werden. Der Grund hierfür ist, dass $\mathcal{T}$ sehr nahe am Merkmal *einfach periodisch* ist.

b)

4.* Die Ausgabe des Werkzeugs zur ratenmonotonen Analyse ist:

```
robge@ipo51:~/txt/job/htw/ezs/src/prak07$ ./tda < tasks6.txt
 Task     t_phi     t_p      t_e     t_d      u_i     t_mr    ok
T(001)    0.00     5.00     1.00     5.00   0.2000    1.00    pass
T(002)    0.00     7.00     1.00     9.00   0.1429    2.00    pass
T(003)    0.00    10.00     3.00    10.00   0.3000    5.00    pass
T(004)    0.00    35.00     7.00    35.00   0.2000   20.00    pass
u = 0.8429, H = 70
```

Um zu ermitteln, wieviel *idle time* noch im Plan ist, kann man einfach die Summe der Längen der einzelnen Jobs von der Länge der Hyperperiode subtrahieren:

$$\begin{aligned} t_{\text{idle}} &= H - \sum_{i=1}^{4} \frac{H}{t_{p,i}} \cdot t_{e,i} \\ &= 70 - \left(\frac{70}{5} \cdot 1 + \frac{70}{7} \cdot 1 + \frac{70}{10} \cdot 3 + \frac{70}{35} \cdot 7 \right) \\ &= 70 - (14 + 10 + 21 + 14) \\ &= 11. \end{aligned}$$

Einfacher wäre, t_{idle} direkt aus der Auslastung abzuleiten:

$$\begin{aligned} t_{\text{idle}} &= H(1 - u) \\ &= H \left(1 - \sum_{i=1}^{4} \frac{t_{e,i}}{t_{p,i}} \right) \\ &= 70 \left(1 - \frac{1}{5} + \frac{1}{7} + \frac{3}{10} + \frac{7}{35} \right) \\ &= 70 \left(1 - \frac{14 + 10 + 21 + 14}{70} \right) \\ &= 70 - 59 = 11. \end{aligned}$$

Für die Taskmenge mit 100 Tasks ist die Bestimmung von H „von Hand“ äußerst mühselig; wir sollten die Ermittlung dem Rechner überlassen. Das kgV der Perioden können wir z. B. effizient mit Hilfe des größten gemeinsamen Teilers der Perioden ermitteln, denn es gilt für zwei beliebige $a, b \in \mathbb{N}$:

$$\text{kgV}(a, b) = a/\text{ggT}(a, b) \cdot b.$$

Der ggT kann effizient mittels des Euklidischen Algorithmus implementiert werden.

Listing 2: C-Code zur Bestimmung von H

```
unsigned long gcd_euclid(unsigned long a, unsigned long b)
{
  if (b==0) {
    return a;
  }
  else {
    return gcd_euclid(b, (a % b)) ;
  }
}

unsigned long lcm(unsigned long a, unsigned long b)
{
  /*TODO: use  __builtin_mul_overflow */

  return a / gcd_euclid(a, b) * b;
}

unsigned long get_hyperperiod(void)
{
  int c;
  unsigned long temp_h;

  temp_h = tasks[0].t_p;
  for (c=1; c<numTasks; c++) {
    temp_h = lcm(temp_h, tasks[c].t_p);
  }
  return temp_h;
}

/*
  We are not sure, whether H is computed correctly, i.e. without
     overflow.
  Therefore, we divide it by every period in the task set and look
     whether
  the result is an integer.

 */
void check_hyperperiod(unsigned long h)
{
  unsigned long di;
  int c;

  for (c=0; c<numTasks; c++) {
    printf("%i %lf\n", c, tasks[c].t_p);
```

```
        di = (h / tasks[c].t_p) * tasks[c].t_p;
        if (di != h) {
          printf("period %lf does not divide H (=%lu). H cannot be
              hyperperiod.\n", tasks[c].t_p, h);
        }
      }
    }
```

Bei sehr vielen verschiedenen Perioden, die zueinander prim sind, birgt dies die Gefahr numerischen Überlaufs, insbesondere, wenn mit 32-Bit-Daten gerechnet wird (z. B. beträgt H für die Datei mit 100 Tasks immerhin 3726084736, was nur knapp kleiner als 2^{32} ist. Dies ist *in praxi* jedoch ziemlich unwahrscheinlich. Der numerische Überlauf bei der Multiplikation könnte durch compilerspezifische Funktionen wie `__builtin_mul_overflow` (bei GCC) behandelt werden, oder, pragmatisch, indem man den ermittelten Wert für H der Reihe nach durch alle Perioden teilt, und auf Ganzzahligkeit des Ergebnisses testet.

Mit $H = 3726084736$ kann man nun für die Taskmenge die Anzahl freier Zeiteinheiten innerhalb von H bestimmen:

$$\begin{aligned} t_{\text{idle}} &= H(1-u) \\ &= 3.726.084.736(1-0.8004) \\ &\approx 743.726.513,3 \end{aligned}$$

Anmerkung: Die Aufgabe ist ein bißchen unsinnig; der Sinn von on-line-Scheduling-verfahren besteht gerade darin, dass *kein* expliziter Plan generiert wird. Im Falle der letzten Taskmenge und einer hypothetischen Dauer einer Zeiteinheit von 1μs würde die Hyperperiode einen Zeitraum von 39 Tagen (!) umfassen; der Platzbedarf für den Plan wäre enorm. Entsprechend ergibt die Angabe der freien Zeiteinheiten innerhalb von H hier keinen Sinn.

5.
- Marssonde „Pathfinder“ landete am 4. Juli 1997 planmäßig auf dem Mars, setzte den Erkundungsrover „Sojourner“ aus und sammelte und sendete fleißig Daten zur Erde.
- Nach einiger Zeit kam es sporadisch zu Resets der Sonde, die Datenverlust nach sich zogen (in der Presse *Software Glitches* genannt)
- Auf der Sonde lief VxWorks, ein kommerzielles RTOS, das Threads präemptiv nach (statischen) Prioritäten plant.
- (u. a.) 3 Tasks:
  - B – „Bus Management Task“: hohe Priorität, hohe Frequenz
  - M – „Meteorological Data gathering Task“: niedrige Priorität, geringe Frequenz

– C – „Communication Task“: mittlere Priorität

B und M griffen auf eine exklusive Ressource, den so genannten „Information Bus“ zu; der Zugriff war durch einen Mutex geschützt.

- Meist funktionierte dies gut. Ab und zu geschah es aber, dass M durch C verdrängt wurde, während es den Mutex besaß. C lief verhältnismäßig lange, B blockierte für diesen Zeitraum, bis ein Watchdog Alarm auslöste und die Sonde wieder per Reset in den Grundzustand überführte. Klassische ungesteuerte Prioritätsinversion geschah.

- Das Problem wurde auf der Erde (im Jet Propulsion Laboratory – JPL – in Pasadena), wo eine exakte Kopie der Sonde stand, analysiert und nach beträchtlichem Aufwand auch reproduziert (der Reset geschah nur ab und zu). Letztlich gelang es den Ingenieuren, mittels einer vergessenen Debug-Schnittstelle, den Initialisierungsparameter des Mutexes so zu modifizieren, dass dieser fortan Prioritätsvererbung beim Blockieren ausführte, und das Problem verschwand dauerhaft.

Interessanterweise war das Problem auch beim Testen sporadisch aufgetreten, doch Termindruck und eine Fokussierung auf den schwierigsten Missionsteil, die Landung, bedingten eine nachlässige Erklärung des Phänomens mit „hardware glitch“.