Aufgrund einer lebendigen Diskussion zum Thema Multitasking in unserem Forum, habe ich mich entschlossen in diesem Artikel einen kleinen, präemptiven Scheduler Schritt für Schritt zu entwickeln und zu erklären.

Grundlegendes

Die Begriffe Multitasking und Multithreading bedeuten, dass von einem Computer mehrere Aufgaben 'quasi' gleichzeitig ausgeführt werden. Bei komplexen Systemen (PCs, Handys, Tablets) nennt man das gleichzeitige Ausführen mehrere Programme Multitasking, die gleichzeitige Ausführung innerhalb eines Programms Multithreading. Auf kleinen Systemen (AVR, kleine ARMs, PIC, 8051) werden beide Begriffe für das selbe verwendet. Dabei müsste man es im engeren Sinn Multithreading nennen, da immer nur ein Programm ausgeführt wird, darin jedoch mehrere C-Funktionen gleichzeitig ausgeführt werden. Beim AVR sind **Threads** und
Tasks
dasselbe und werden nebenbei auch noch als
Prozesse
bezeichnet - drei Wörter für ein Ding!

Der AVR-Prozessor kann jedoch immer nur eine Sache gleichzeitig bearbeiten. Die 'quasi' gleichzeitige Ausführung erreicht man dadurch, dass man immer zwischen den Verschiedenen Aufgaben hin und herwechselt.

Beispiel: Fünf Zeilen lesen, an der Kaffetasse nippen, gucken ob man eine neue Mail hat, fünf weitere Zeilen lesen, an der Kaffetasse nippen, gucken ob man eine neue Mail hat, ...

Für das Umschalten hat man die beiden Möglichkeiten Automatisch und Manuell. Das automatische Umschalten wird meistens mit einem Timer-Interrupt gemacht. Es kann zum Beispiel alle 10 ms die aktuelle Aufgabe unterbrochen werden und zu einer anderen Aufgabe gewechselt werden. Die automatische Umschaltung nennt man **präemptives Multitasking**.

Die manuelle Umschaltung bezeichnet man mit **kooperativem Multitasking**. Kooperativ bedeutet das man in seinen Funktionen regelmäßig eine Yield() Funktion aufruft. Beim Aufruf der Yield Funktion werden die anderen Prozesse abgearbeitet. Yield() bedeutet also 'mach kurz

was anderes und mach dann hier weiter'.

Beispiel kooperatives Multitasking:

```
  void threadMain1 () {
  while (true) {
    leseZeilen(5);
    thread_yield();
  }
}

void threadMain2 () {
  while (true) {
    nippKaffee();
    thread_yield();
  }
}

void threadMain3 () {
  while (true) {
    pruefeMails();
    thread_yield();
  }
}
```

Wie sieht das Multitasking aus Sicht des Programms aus?

Globale Variablen können von allen Prozessen gemeinsam benutzt werden. Dazu müssen sie dann **volatile** definiert werden, sonst kann es Probleme geben!
Lokale Variablen der jeweiligen Thread-Funktion und die lokalen Variablen der aufgerufenen Funktionen gehören nur zu ihrem jeweiligen Prozess.

Beispiel:

```
  volatile unsigned int zaehler; // globale Variable, Zugriff von allen Threads möglich

void threadMain1() {
  while (true) {
    zahler++;
  }
}
```

```
void threadMain2() {
  unsigned int zahler2 = 0; // lokale Variable, Zugriff nur von threadMain2 möglich
  while (true) {
    if (zahler>100) {
      zahler2++;
      zaehler = 0;
    }
  }
}
```

Wie können sich die verschiedenen Tasks miteinander verständigen?

Die Verständigung findet einmal über globale Variablen statt zum anderen über ein sogenannte Signale. Ein Prozess kann an einen anderen Prozess ein Signal senden - der andere Prozess kann auf sein Signal warten.

Beispiel:

```
  // in threadMain1:
thread_wait ();
// in threadMain2:
thread_signal (&thread1);
```

Der wartende Prozess wird solange nicht mehr durch den Scheduler aufgerufen, bis ihm ein Signal gesendet wird.
Es existieren noch viele weitere Synchronissationsmechanismen (Mutex, Semaphor), die sich jedoch alle auf Signale und globale Variablen zurückführen lassen.

Was benötigen wir für den Anfang?

Zunächst wollen wir uns mit einem kooperativen Multitasking zufrieden geben. Später fügen wir dann die Signale und das preemptive Verhalten hinzu.

Für die Implementation ist es zunächst wichtig zu erkennen woraus überhaupt ein Prozess besteht:

- Die aktuelle Position im Programmcode
- Der Weg der Funktionsaufrufe dorthin
- Der Zustand der lokalen Variablen aller aktiven Funktionen

Diese Informationen nennt man den **Kontext** des Prozesses.

Es ist also nicht nur wichtig welche Funktion gerade aufgerufen wird, und an welcher Stelle man sich befindet, sondern auch von welcher anderen Stelle die Funktion aufgerufen wurde!
Beispiel:

```
  funktionA() {
}
funktionB() {
}
threadMain() {
  funktionA();
  funktionB();
  funktionA();
}
```

Wenn wir uns in der Funktion A befinden ist es wichtig zu wissen ob wir von der ersten oder der zweiten Stelle aus aufgerufen worden sind, damit wir nach der Fertigstellung der Funktion zum richtigen Punkt zurückkehren können.

Da dieses jedoch auch ganz normale C-Programme schon können brauchen wir uns um die Details keine Gedanken zu machen, wir müssen nur wissen das der genaue Zustand unseres Prozesses auf dem Stack und in den Registern gespeichert ist. Beim umschalten zwischen verschieden Prozessen können wir den Inhalt der Register auch auf den Stack schreiben und von dort laden - dann sind praktischer Weise alle Daten des Prozesses nur noch auf dem Stack!!

1. Erkenntnis: Jeder Prozess benötigt einen eigenen Stack!

Exkurs: Verkettete Liste

Um unser System zu organisieren benötigen wir noch eine Datenstruktur: Eine *zirkuläre doppelt verkettete Liste mit Kopf*
(
de.wikipedia.org
). Der Begriff ist etwas sperrig ;-)

Wenn wir unsere Prozesse mit der Liste organisieren, hat jeder Prozess einen Vorgänger und einen Nachfolger (doppelt verkettet). Durch 'abwandern' der Liste kann man alle Prozesse erreichen und kommt am Ende da raus wo man angefangen hat (zirkulär). Der zusätzliche Kopf macht vieles einfacher , da bei einer leeren Liste der Kopf trotz alledem enthalten ist und dabei sein eigener Vorgänger und Nachfolger ist. Somit gibt es keine Sonderfälle beim Löschen und Einfügen...

Die Datenstruktur:

```
  typedef struct _cdll_s
{
   struct _cdll_s * prev;
   struct _cdll_s * next;
} cdll_t;
```

Daneben benötigen wir noch einige Funktionen zur Verwaltung der verketteten Listen:

```
  void cdll_reset(cdll_t *head);
bool cdll_is_empty(cdll_t *head);
void cdll_push_front(cdll_t * head, cdll_t * entry);
void cdll_push_back(cdll_t * head, cdll_t * entry);
cdll_t * cdll_pop_front(cdll_t * head);
cdll_t * cdll_pop_back(cdll_t * head);
void cdll_remove(cdll_t * entry);
bool cdll_contains(cdll_t * head, cdll_t * entry);
```

Die Liste selbst (der *head*) und die Elemente haben bei dieser Form alle denselben Typ: **cdll_t**

Die Dateien für die CDLL (Circular Double Linked List - zirkuläre doppelt verkettete Liste):

- cdll.h
- cdll.c

Exkurs: Stack

Alle Daten die zu unserem Prozess (Thread/Task) gehören werden auf einem Stack gespeichert. In unserem Stack können Bytes abgespeichert werden. Wie bei einem Papierstapel auf dem Schreibtisch kann man immer nur auf das obere Ende des Stapels zugreifen (wühlen ist verboten!). Man kann entweder ein weiters Blatt (Byte) drauflegen oder ein Blatt runternehmen.

Der in den Atmel AVRs eingebaute Stack arbeitet in genau umgekehrte Richtung (sozusagen ein Tiefstapler...). Er schreibt den ersten Wert an die Endaddresse des Stackbereichs und arbeitet sich von dort in Richtung Anfang vor. Ein spezielles Register in der CPU - der Stackpointer SP - zeigt immer auf die nächste freie Adresse. Der genaue Bezeichnung für dieses Verhalten ist "Post-Dekrement-Push-Stack".

Die beiden Operationen um auf den Stack zuzugreifen lauten:

- PUSH - ein Byte ablegen (C: *byte_ptr-- = wert)
- POP - ein Byte zurückholen (C: wert = *++byte_ptr)

Multitasking Teil 2

Quellen:

Als Quelle der Inspiration und vieler Details dienten folgende Systeme:

- XMK - www.shift-right.com/xmk
- tnKernel - www.tnkernel.com
- GOS - www.microcontroller.net
- Wikipedia - de.wikipedia.org