Aufgabe 3: QNX - Verarbeitungszeit

Embedded Computing, Teilgruppe 2, Florian Tobusch - Chris Brammer

Implementierung von waste_msecs(unsigned int msecs)

Um Zeit zu verbrauchen wird die CPU damit beschäftigt die Fibonacci Zahlen zu berechnen. Dies geschieht in einer Schleife welche über eine globale Variable `waste_scaling` mehr oder weniger oft durchlaufen wird.

```
void waste_msec_internal() {
    volatile int first = 0, second = 1, next, c;
    for (c = 0; c < (int) (200000 * waste_scaling); c++) {
        if (c <= 1) {
            next = c;
        }
    else {
            next = first + second;
            first = second;
            second = next;
        }
    }
}
```

Um `waste_msecs(unsigned int msecs)` zu realisieren muss `waste_scaling` so gesetzt werden das die Berechnung 1 Millisekunde dauert. Dann kann in `waste_msecs` die Anzahl der zu verbrauchenden Millisekunden eingestellt werden.

```
void waste_msec(unsigned int msecs) {
    int i;
    for (i = 0; i < msecs; i++)
        waste_msec_internal();
}
```

Kalibrierung

Gemessen werden eine bestimmte Anzahl Iterationen des Fibonacci Aufrufs (`waste_msec_internal`). Da jeder Aufruf genau 1ms dauern soll wird der Skalierungsfaktor `waste_scaling` auf den inversen Wert der gemessen Zeit pro Aufruf gesetzt.

```
if (-1 == clock_gettime(CLOCK_REALTIME, &current)) {
  perror("clock_gettime");
}

waste_msec(ITERATIONS);
```

```
if (-1 == clock_gettime(CLOCK_REALTIME, &after)) {
  perror("clock_gettime");
}

float time_slept_ms = 0;
time_slept_ms += (after.tv_sec - current.tv_sec) * 1000.f;
time_slept_ms += (after.tv_nsec - current.tv_nsec) / (1000.f * 1000.f);
time_slept_ms /= ITERATIONS;

waste_scaling = 1.f / time_slept_ms;
```

Genauigkeit

Bei Wartezeiten von 1ms bis 2000ms auf dem Main-Thread wurden Abweichungen im Rahmen von 0-10ms gemessen. Es gab hierbei keine Korrelation zur Wartezeit selbst.

Prioritäten laufender Prozesse

Alle laufende Prozesse und Informationen zu diesen kann über den Befehl pidin (http://www.qnx.com/developers/docs/7.0.0/#com.qnx.doc.neutrino.utilities/topic/p/pidin.html) abgefragt werden. Hierfür muss eine Verbindung über `telnet` auf das BeagleBone hergestellt werden. Es wird ersichtlich das 0 die niedrigste Priorität ist.

Erstelung eines Threads mit gehobener Priorität

```
pthread_t worker;
pthread_attr_t worker_attr;

struct sched_param worker_sched_param; //sched_curpriority is ignored
worker_sched_param.sched_priority = sched_get_priority_max(SCHED_RR);

pthread_attr_init(&worker_attr); //initialize thread-attribute object to default va
lues
pthread_attr_setinheritsched(&worker_attr, PTHREAD_EXPLICIT_SCHED); //disable sched
uling policy from parent thread
pthread_attr_setschedparam(&worker_attr, &worker_sched_param); //define scheduling
 parameters

if (EOK != pthread_create(&worker, &worker_attr, &worker_thread, NULL)) {
    perror("Error during thread creation");
}

pthread_join(worker, NULL);
```

Messungen auf dem Thread

Gleich wie auf dem Main-Thread wird auf dem Thread zuerst eine Kalibrierung durchgeführt und anschliessend eine Zeit gewartet. Die tatsächlich vergangene Zeit wird gemessen.

Klar zu sehen war das durch die hohe Priorität des Threads die Ergebnisse sehr viel konsistenter sind da andere Prozesse (Single Core) uns nicht unterbrechen können. Zum Beispiel kommt der Debugger (Prozess auf dem Board) nicht mehr zum Zuge um z.B Ausgaben an die IDE zu schicken.

Einfluss des System Ticks

Der System Tick wurde mit 1ms gewählt da dieser einen großen Einfluss auf die Ausführungszeit der Messung haben kann. Da der System Tick das System unterbricht (unabhängig von der Priorität) wird bei steigender Tickfrequenz `waste_msecs` öfter unterbrochen und somit die Ausführungszeit gestreckt.