Vorlesung Betriebssysteme WS 2018/2019

Aufgabenblatt 4 vom 11. Januar 2019

(Vorstellung der Lösungen bei den Tutoren bis zum 25.01.2019)

1. Beantworten Sie die folgenden Fragen und erläutern Sie die Antworten Ihrem Tutor.
   (a) Was ist *busy waiting*?
   (b) Welche anderen Möglichkeiten zu warten gibt es für Threads in einem Betriebssystem?
   (c) Kann *busy waiting* immer vermieden werden?
   (d) Welche Synchronisationsmechanismen im Windows-Kernel nutzen *busy waiting*? Was können Sie über deren Implementierung im WRK[1] sagen?

2. Untersuchen Sie den Programmrahmen `multithread.zip` auf der Vorlesungsseite. Die darin enthaltene Datei `multithread.c` berechnet ein fraktales Bild als BMP-Datei; jedoch bisher nur sequentiel.

   Erweitern Sie das Programm, um das fraktale Bild unter Windows und Unix beschleunigt mithilfe mehrerer Threads zu berechnen. Identifizieren Sie kritische Abschnitte und lösen Sie diese durch Synchronisationsmechanismen und/oder Umformulierung des Programms auf.

   Die Anzahl der Threads, die zur Berechnung verwendet werden, soll als erster Parameter der Programmaufrufs erwartet werden. Beispiel:

   ```
   $ ./multithread 65
   ```

   Ihr Programm soll mindestens 65 Threads unterstützen. Auftretende Fehler sollen von Ihnen korrekt behandelt werden. Verpacken Sie alle Quellcode-Dateien mit Makefiles in ein ZIP-Archiv und laden Sie dieses Archiv mindestens 24 Stunden vor dem Tutoriumstermin ins Abgabesystem hoch.

[1] `http://wrk.dcl.hpi.uni-potsdam.de`

3. Gegeben sei ein Einprozessor-System welches Round-Robin-Scheduling mit 16 Prioritätsstufen verwendet (0-15, 0 = niedrigste, 15 = höchste Priorität). Die Quantumslänge beträgt 20ms. Die Zeit für einen Kontextwechsel sei vernachlässigbar. Der Scheduler verwaltet laufende Threads und entscheidet ausschließlich nach dem Ablauf eines Quantums welcher Thread als nächstes laufen soll. Das Einfügen in die Warteliste erfolgt nach FIFO-Ordnung.

   Es sollen drei Threads mit den folgenden Eigenschaften ausgeführt werden:

Thread ID	Startzeit t_s (in ms)	Ausführungszeit e (in ms)
1	0	100
2	15	80
3	30	60

   Erstellen Sie ein Gantt-Diagramm unter der Annahme, dass alle drei Threads mit einer (statischen) Priorität von 8 ausgeführt werden.

   Erstellen Sie ein zweites Gantt-Diagramm. Hier soll Th3 mit einer Priorität von 9 ausgeführt werden. Weiterhin erhält Th1 eine Priorität von 7 und tritt nach 16ms in einen I/O-Wartezustand. Die Priorität von Th1 soll nach Beendigung der I/O-Operation um drei erhöht werden (*Boost*). Th1 verlässt den Wartezustand bei t=45ms. Die Priorität von Th1 wird um eine Stufe am Quantumsende reduziert, bis wieder die Basispriorität erreicht ist. Zeichnen Sie ein Gantt-Diagramm für den beschriebenen Vorgang.

   Ergänzung: Die Ordinatenachse Ihres Diagramms sollte die aktuelle Threadpriorität abbilden.

4. Zwei parallel ablaufende Prozesse (der *Erzeuger* und der *Verbraucher*) verwenden einen gemeinsamen, begrenzten Speicherbereich. Der Erzeuger schreibt Daten in den gemeinsamen Speicher, der Verbraucher entnimmt Daten. Kann der Erzeuger kein neues Datenelement speichern, verwendet er die `suspend`-Funktion und wartet, bis er vom Verbraucher wieder aufgeweckt wird, nachdem dieser ein Datenelement entfernt hat. Analog wartet der Verbraucher – wenn keine Datenelemente vorhanden sind – bis er vom Erzeuger geweckt wird.

   Nachfolgend sind (Pseudo-Code-)Implementierungen für den Erzeuger und den Verbraucher dargestellt:

```
#define N 100  // Puffergroesse
int count = 0; // Anzahl der Puffereintraege
```

```
void producer (void)
{
  int item;

  while (True) {
      produce_item(&item);
      if (count == N)
        suspend();
      insert_item(item);
      count = count + 1;
      if (count == 1)
        wakeup(consumer)
    }
}
```

```
void consumer (void)
{
  int item;

  while (True) {
      if (count == 0)
        suspend();
      remove_item(&item);
      count = count - 1;
      if (count == N - 1)
        wakeup(producer);
      consume_item(item);
    }
}
```

Analysieren Sie das Programmfragment und beantworten Sie die folgenden Fragen:

(a) Da der Erzeuger- und Verbraucherprozess parallel ablaufen, können sie vom Scheduler unterbrochen werden. Finden Sie zwei unterschiedliche parallele Programmabläufe, die dazu führen, dass sowohl der Erzeuger als auch der Verbraucherprozess mittels `suspend` warten (*Deadlock*).

(b) Warum wird dieses Problem nicht gelöst, wenn man die gesamten `while`-Schleifenrümpfe als kritische Abschnitte schützt?

(c) Beschreiben und begründen Sie, wie dieses Problem mithilfe von Semaphoren gelöst werden kann.