Vorlesung Betriebssysteme

Prof. Dr. Jan Dünnweber

Verteilte Systeme und Betriebssysteme
Fakultät für Informatik und Mathematik
Ostbayerische Technische Hochschule Regensburg

Wintersemester 2013/14

Logging für Pthreads

- In dieser Übung sollen die Effekte unterschiedlicher **Scheduling Strategien** analysiert werden
- Vereinbaren Sie zunächst Datenstrukturen für das *Logging*:

Die folgenden Vereinbarungen sollen *global* gelten

```
#define THREADS 5
#define STEPS_PER_ACTIVITY 5
pthread_mutex_t lock;  /* access to log is exclusive */
struct timestamp {  int id, value;  }
*logData[THREADS * STEPS_PER_ACTIVITY];  int lineCount;
```

- Zudem wird eine Funktion benötigt, die zwei Log-Einträge gemäß ihrer **Response Time** vergleicht:

Vergleichsfunktion für `qsort`

```
int timeCmp(const void *v1, const void *v2)  {
    struct timestamp *ts1 = (struct timestamp *)v1,
                     *ts2 = (struct timestamp *)v2;
    return ts1->value - ts2->value;
}
```

Aktivitäten mit Zeitstempeln

- Die Aufgabe (`task`) der einzelnen Threads besteht für diese Übung ausschließlich darin, ihre `IDs` mit Zeitstempeln in dem zuvor vereinbarten `log`-Array zu speichern

Code für die "Aktivitäten"

```
void *task(void *data)  {
   int i, nr = *((int *)data);
   for (i = 0;  i < STEPS_PER_ACTIVITY;  ++i)  {
      pthread_mutex_lock(&lock);    /* enter critical section */
      struct timestamp *value = (struct timestamp *)
                      malloc(sizeof(struct timestamp));
      struct timespec value;
      clock_gettime(CLOCK_REALTIME, &value);
      stamp->id = nr;
      stamp->stamp = value.tv_nsec;
      logData[lineCount++] = stamp; /* write log data to shared memory */
      pthread_mutex_unlock(&lock);  /* leave critical section */
   }
}
```

Analyse des Ablaufs

- Ein konkreter Ablauf lässt sich mit den zuvor getroffenen Vereinbarungen wie folgt analysieren:

Code zur Ausgabe eines Ablaufprotokolls

```
void analyzeScheduling(pthread_attr_t *ap)  {
   int i, n[THREADS];  pthread_t t[THREADS];
   lineCount = 0;
   for (i = 0;  i < THREADS;  ++i)  {
      n[i] = i;           /* start new thread with id = i */
      pthread_create(&t[i], ap, task, &n[i]);
   }
   for (i = 0;  i < THREADS;  ++i)
      pthread_join(t[i], NULL); /* let all threads finish */
   /* sort log entries according to their timestamps:     */
   qsort(logData, lineCount, sizeof(struct timestamp *), timeCmp);
   for (i = 0;  i < lineCount;  ++i)
      printf( "%2d: Thread #%d performed an activity\n",
               i + 1, logData[i]->id );
}
```

Thread Contention

- Erweitern Sie die Quelltexte von den vorigen Folien um ein Hauptprogramm (`main`), das mehrere Abläufe für *unterschiedliche* Scheduling Strategien protokolliert

  Hinweise:

  - Verwenden Sie `pthread_attr_setschedpolicy` für die Auswahl der Strategie
  - Beachten Sie, dass `pthread_attr_init(&attr)` und `pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED)` aufgerufen werden müssen, damit das Betriebssystem Ihre Auswahl berücksichtigt
  - Um beim Zugriff auf die Log-Daten wechselseitigen Auschluß zu gewährleisten, muss auch die Mutex-Variable (`lock`) initialisiert werden (mit `pthread_mutex_init(&lock)`)
- Verwenden Sie `pthread_attr_setscope` und untersuchen Sie die Auswirkungen von *Thread Contention* auf Prozessebene (bzw. System-weit)