Vorbereitung

- Die Dateien zum heutigen Praktikumstermin laden Sie von der Vorlesungsseite herunter:
  `wget http://hm.hgesser.de/bs-ss2009/prakt05.tgz`
- Entpacken Sie das Archiv und wechseln Sie in das neue Unterverzeichnis `prakt05`.

10. Round-Robin- und VRR-Scheduler

Aus dem Quellcode-Archiv entnehmen Sie die Datei `sched.py`, die einen FCFS-Scheduler implementiert. (Es handelt sich um eine leicht angepasste Variante des Scheduler-Programms aus Aufgabe 9.)

a) Verwenden Sie diese Datei als Ausgangsbasis für einen Round-Robin- (RR-) Scheduler – FCFS und RR unterscheiden sich nur dadurch, dass RR nach Ablauf eines vorgegebenen Zeitquantums den laufenden Prozess unterbricht und ans Ende der Warteschlange anhängt.

   Beachten Sie für Ihre Lösung die folgenden Punkte:

   - Die Länge des Quantums sollte sich über eine Variable einstellen lassen; alternativ lesen Sie über `argv[2]` ein zweites Argument aus, das der Scheduler als Länge des Quantums interpretiert.
   - Damit Prozesse nie mehr Zeit als das Quantum verbrauchen, erweitern Sie den Prozesskontrollblock (also jeweils das Dictionary `tasks[i]`) um einen Zähler. Prüfen Sie, wann genau Sie diesen Zähler zurück auf 0 setzen müssen – neben den offensichtlichen Fällen (Initialisierung und Entzug der CPU durch den Scheduler) gibt es noch eine andere Situation. Der zur Verfügung gestellte Quellcode enthält bereits Funktionen zum Auslesen, Schreiben und Erhöhen dieses Zählers `usedquant`.

b) In der Vorlesung haben Sie gesehen, dass der Standard-RR-Scheduler CPU-lastige Prozesse bevorzugt. Die Variante Virtual Round Robin (VRR) behebt diese ungleiche Behandlung von I/O- und CPU-lastigen Prozessen. Erzeugen Sie eine Kopie Ihres RR-Schedulers, z. B. `sched-vrr.py`, und ergänzen Sie die bevorzugte Warteschlange, in die Prozesse wechseln, die eine I/O-Operation starten, ohne ihr volles Quantum aufgebraucht zu haben.

c) Testen Sie beide Scheduler mit dem Beispiel aus der Vorlesung, das wir dort zur Unterscheidung von RR und Virtual RR verwendet haben – prüfen Sie, ob Ihr VRR-Scheduler wirklich I/O-lastige Prozesse besser stellt als der RR-Scheduler.

d) Um die für Context Switches benötigte Rechenzeit mit in die Untersuchungen einzubeziehen, erweitern Sie den RR-Scheduler aus Aufgabenteil a) um einen Context-Switch-Zähler. Jedesmal, wenn der Scheduler die Entscheidung trifft, einen anderen Prozess auszuwählen, oder wenn wegen des Wechsels in einer I/O-Phase ein anderer Prozess an die Reihe kommt, soll dieser Zähler erhöht werden. Außerdem soll dann auch die Uhr hochgezählt werden (nehmen Sie an, dass ein Context Switch eine Zeiteinheit benötigt), so dass die Context-Switch-Zeiten auch in den Gesamtrechenzeiten der Prozesse auftauchen.

11. POSIX-Threads und Semaphore in C

Lesen Sie, ausgehend von der Manpage `sem_overview`, die Dokumentation zu Semaphoren unter Linux und betrachten Sie dann das Programm `reader-writer.c`, das eine ausformulierte Fassung des Producer-Consumer-Beispiels aus der Vorlesung (Foliensatz 5, Folie 43) ist.

Auf der folgenden Seite finden Sie die wichtigsten Ausschnitte aus dem Code:

```
#define N 10                  // Groesse des Puffers

// Semaphor-Definitionen, siehe: man sem_overview
static sem_t mutex;           // kontrolliert Zugriff auf Puffer
static sem_t empty;           // zaehlt freie   Plaetze im Puffer
static sem_t full;            // zaehlt belegte Plaetze im Puffer

// Hauptprogramm des Producer-Prozesses -- wird als neuer Thread gestartet
void producer() {
  char item;
  while (item != NOCHAR) {
    item = produce_item();    // Erzeuge etwas für den Puffer
    sem_wait (&empty);        // Leere Plaetze dekrementieren bzw. blockieren
    sem_wait (&mutex);        // Eintritt in den kritischen Bereich
    enter_item (item);        // In den Puffer einstellen
    sem_post (&mutex);        // Kritischen Bereich verlassen
    sem_post (&full);         // Belegte Plaetze erhoehen, evtl. Consumer wecken
  }
}

// Hauptprogramm des Consumer-Prozesses -- wird als neuer Thread gestartet
void consumer() {
  char item;
  while (item != NOCHAR) {
    sem_wait (&full);          // Belegte Plaetze dekrementieren bzw. blockieren
    sem_wait (&mutex);         // Eintritt in den kritischen Bereich
    item = remove_item();      // Aus dem Puffer entnehmen
    sem_post (&mutex);         // Kritischen Bereich verlassen
    sem_post (&empty);         // Freie Plaetze erhoehen, evtl. Producer wecken
    consume_entry (item);      // Verbrauchen
  }
}

// Hauptprogramm
main() {
  pthread_t consumer_thread;  // Thread-Variablen fuer Consumer ...
  pthread_t producer_thread;  // ... und Producer
  sem_init(&mutex, 0, 1);     // Init: 1, Wertebereich: 0-1 (Mutex)
  sem_init(&empty, 0, N);     // Init: N, Wertebereich: 0-N
  sem_init(&full,  0, 0);     // Init: 0, Wertebereich: 0-N

  init_buffer(); printf ("Startwerte:\n"); list();
  // Threads erzeugen
  pthread_create( &producer_thread, NULL, (void*)&producer, NULL);
  usleep(100);
  pthread_create( &consumer_thread, NULL, (void*)&consumer, NULL);

  sleep(1); printf("\n\nEndwerte:\n"); list();
  // Aufraeumen:
  pthread_join( producer_thread, NULL ); pthread_join( consumer_thread, NULL );
}
```

Sie übersetzen das Programm mit

```
gcc -o reader-writer reader-writer.c -lpthread
```

und können es dann ausprobieren. Die `usleep`-Aufrufe in den beiden Threads dienen dazu, dem jeweils anderen Thread eine Chance zu geben, den Mutex zu erhalten.

a) Modifizieren Sie das Programm, so dass es mit beliebig vielen Erzeuger- und Verbraucher-Threads arbeitet – wie viele, legen dann zwei Variablen (oder Konstanten) `NO_READERS` und `NO_WRITERS` im Programm fest.

b) Verändern Sie das Hauptprogramm und die Funktion `list()`, so dass `list()` in eine Protokolldatei schreibt und regelmäßig aus dem Haupt-Thread heraus `list()` aufgerufen wird. Prüfen Sie, wie sich der Puffer füllt.

[☛ Abgabe per Mail] Die Lösungen schicken Sie mir bitte per Mail, `hans-georg.esser@hm.edu`.