Musterlösungen zu den Scheduler-Programmierübungen 06.07.2011, Hans-Georg Eßer

Übung 7

18 a)

Die Ausführreihenfolge, die der Python-Scheduler anzeigt, entspricht der Reihenfolge, die in der Vorlesung präsentiert wurde.

18 b)

- `create_process` wird in `init()` für jeden Prozess in der Prozesskonfigurationsdatei aufgerufen und trägt ihn in die Datenstruktur `tasks` ein. Dabei wird auch der Sonderfall überprüft, dass das Verhalten mit 0 anfängt, der Prozess also mit einer I/O-Phase startet. Abhängig von dieser Prüfung wird der Status initial auf `S_READY` oder `S_BLOCKED` gesetzt. Das Feld `firstruntime` wird auf -1 gesetzt, weil kein Prozess zu diesem Zeitpunkt bereits gelaufen ist.
- `run_current` vermerkt beim ersten Aufruf für einen Prozess die `firstruntime` im zugehörigen Feld. Die verbleibende Laufzeit in `behavior` wird um 1 reduziert. Wird dabei der Wert 0 erreicht, ist der aktuelle CPU Burst zu Ende, und der Prozess wechselt in den Zustand `S_BLOCKED` und wird aus der `runqueue` entfernt. Falls aber keine Blocked-Phase mehr folgt (und der Prozess fertig ist), wird der Status auf `S_DONE` gesetzt und die Endzeit in PCB vermerkt. Anderenfalls wird der Prozess in die `blocked`-Warteschlange eingetragen.
- `update_blocked_processes` verringert zunächst für Prozesse in der `blocked`-Warteschlange die Wartezeit, aber nur, wenn sie bereits „im System sind“ - diese Einschränkung fängt den Sonderfall ab, dass ein Prozess mit einer I/O-Phase startet und sein Startzeitpunkt noch nicht erreicht ist, denn er taucht dann bereits in der `blocked`-Warteschlange auf. Erreicht die Wartezeit den Wert 0, wird analog zu `run_current` geprüft, ob der Prozess fertig ist (dann Status auf `S_DONE` setzen, Endzeit eintragen) oder ob noch eine CPU-Phase folgt (dann Status auf `S_READY` setzen und Prozess in `runqueue` eintragen; in beiden Fällen wird der Prozess aus der `blocked`-Warteschlange entfernt.

18 c)

SJF ist – wie FCFS – ein nicht-unterbrechender, also kooperativer Scheduler. Das Grundprinzip, dass ein bereits laufender Prozess weiterläuft, bleibt also erhalten.
Abweichend von FCFS wird nach Fertigstellung eines Prozesses geprüft, welcher der bereits in der Warteschlange stehenden Prozesse den kürzesten nächsten CPU-Burst hat; dieser Wert steht für jeden Prozess aus `runqueue` in `behavior[0]`.
Eine einfache Lösung wählt zunächst den ersten Prozess in `runqueue` und vergleicht dann mit allen übrigen, die Befehle

```
# falls nicht: nehme ersten Prozess aus Runqueue
if (runqueue != []):
  return runqueue[0]
```

in der Funktion `schedule()` werden also ersetzt durch:

```
if (runqueue != []):
  choice = runqueue[0]
  minburst = choice["behavior"][0]
  for p in runqueue[1:]:
     tmp = p["behavior"][0]
     if tmp < minburst:
        choice = p
        minburst = tmp
  return choice
```

und der Rest bleibt wie gehabt.

18 d)
Der SRT-Scheduler verwendet für die Auswahl eines zu aktivierenden Prozesses dieselbe Logik wie SJF (was die Suche angeht). Allerdings ist er unterbrechend, schaut also nach jedem Schritt, ob es inzwischen einen (neuen) Prozess gibt, der einen kürzeren nächsten CPU Burst hat.
Um den SJF-Scheduler in einen SRT-Scheduler umzuwandeln, lässt man in `schedule()` einfach am Anfang die Überprüfung weg, ob der der aktive Prozess noch weiter laufen kann, und geht direkt zur Suche nach dem kürzesten über. Man entfernt also die Zeilen

```
  # falls aktueller Prozess noch bereit: weitermachen
  if (current!=-1) and (get_status(current) == S_ACTIVE):
    return current
```

Übung 8

19 a)
Die Musterlösung zum RR-Scheduler liegt in der Datei `sched-rr.py`. Wesentliche Änderungen gegenüber dem FCFS-Scheduler sind hier:

- Bei der Initialisierung eines Prozesses einen Zähler `usedquant` anlegen und auf 0 setzen (schon in Datei aus Aufgabenstellung vorgegeben, in `create_process`)
- In `run_current(): set_usedquant(current, get_usedquant(current)+1 )` einfügen, um das genutzte Quantum zu erhöhen. Wenn die CPU-Phase zu Ende ist (und der Prozess nach `blocked` verschoben wird), auch `usedquant` auf 0 setzen.
- In `schedule()`: prüfen, ob `usedquant` bereits das maximale Quantum erreicht hat – wenn ja, diesen Prozess nicht weiter ausführen, sein `usedquant` auf 0 zurücksetzen und Prozess in die Warteschlange schieben. Aufpassen bei der Fallunterscheidung in `schedule()`, dass in allen Fällen ein vernünftiger Wert für `choice` ermittelt und auch zurückgegeben wird.

19 b)
Die Musterlösung zum VRR-Scheduler (Virtual Round Robin) liegt in der Datei `sched-vrr.py`. Wesentliche Änderungen gegenüber `sched-rr.py` sind:

- Verwenden einer zusätzlichen Ready-Queue namens `priority`. Sie enthält die Prozesse, die bei der letzten I/O-Blockade noch nicht ihr Quantum aufgebraucht hatten und wieder bereit geworden sind. Der Scheduler soll sie bevorzugt dran nehmen, wenn es zur Auswahl eines neuen Prozesses kommt.
- In `update_blocked_processes()`: Beim Eintragen in die Ready-Queue stand hier vorher nur `runqueue.append(pid)`. Jetzt Auswahl, ob Prozess in die normale oder in die `priority`-Queue eingetragen wird (basierend auf `usedquand`)
- In `schedule()`: Bei allen Tests auf weitere vorhandene Prozesse auch `priority` berücksichtigen. Bei Auswahl des Prozesses zunächst in `priority` gucken
- Die Liste `priority` muss in den beteiligten Funktionen auch als globale Variable definiert werden.
- Hier darf in `run_current` nicht `usedquant` auf 0 gesetzt werden, wenn ein Prozess blockiert!

19 c)
Für Tests eignet sich folgende Prozessdatei `test-vrr`.dat:

```
0:2,2,2,2,2,2,-1
0:10,2,10,-1
0:10,2,10,-1
```

Der erste Prozess (PID 0) läuft immer nur kurz, bevor die nächste I/O-Phase kommt. Als Ausführreihenfolge ergibt sich damit

```
Trace: [0, 0, 1, 1, 1, 1, 1, 0, 0, 2, 2, 2, 2, 2, 0, 1, 1, 1, 1, 1, 2, 2, 2, 2,
2, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 'END']
```

Der Prozess 0 läuft also priorisiert, zunächst 2x, dann 2x, dann nur 1x (VRR!) Dieses 1x-Laufen folgt aus der VRR-Berücksichtigung des Restquantums.

19 d)
Die Musterlösung zum RR-Scheduler mit Context-Switch-Zähler liegt in der Datei `sched-rr-switch.py`. Wesentliche Änderungen gegenüber `sched-rr.py` sind:

- in `schedule()`: Aufruf einer neuen Funktion `contextswitch()`, falls ausgewählter Prozess (`choice`) nicht mit bereits laufendem identisch ist (`current`). Das deckt auch den Fall ab, in dem alle blockieren. Hier kann man diskutieren, ob der Übergang „x läuft“ → „alle blockiert“ → „y läuft“ zwei Context Switches oder einen darstellt; in meiner Musterlösung sind es zwei Switches.
- neue Funktion `contextswitch()`: Hochzählen der CPU-Zeit, Eintrag in `trace`

Alle Dateien liegen in `scheduler-loesungen.zip`.