Stephan Brumme, SST, 3.FS, Matrikelnr. 70 25 44
mail@stephan-brumme.com

Aufgabe 1

Die nachfolgenden Tabellen sind bei einem Hit dunkelgrau, bei einem Miss hellgrau markiert. Die Zeile „Seitenfehler“ ist ein fortlaufender Zähler aller Misses. Je niedriger dieser Wert am Ende der Seitenreferenzfolge ist, desto effizienter arbeitet der Algorithmus.

a) Least-Recently-Used (LRU):
Es wird diejenige Seite verdrängt, auf die am längsten nicht mehr zugegriffen wurde.

Seitenreferenzfolge	3	1	5	8	8	1	2	4	3	4	2	5	6	7	1	1	5
Seitenrahmen 1	3	3	3	3	3	3	2	2	2	2	2	2	2	2	1	1	1
Seitenrahmen 2		1	1	1	1	1	1	1	1	1	1	5	5	5	5	5	5
Seitenrahmen 3			5	5	5	5	5	4	4	4	4	4	4	7	7	7	7
Seitenrahmen 4				8	8	8	8	8	3	3	3	3	6	6	6	6	6
Seitenfehler	1	2	3	4			5	6	7			8	9	10	11		

b) Least-Frequently-Used (LFU):
Es wird diejenige Seite verdrängt, auf die am seltensten zugegriffen wurde. Gibt es mehrere Seiten mit gleicher Zugriffshäufigkeit, so kommt LRU zum Zuge.

Seitenreferenzfolge	3	1	5	8	8	1	2	4	3	4	2	5	6	7	1	1	5
Seitenrahmen 1	3	3	3	3	3	3	2	2	3	3	2	5	6	7	7	7	5
Seitenrahmen 2		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Seitenrahmen 3			5	5	5	5	5	4	4	4	4	4	4	4	4	4	4
Seitenrahmen 4				8	8	8	8	8	8	8	8	8	8	8	8	8	8
Zähler für S. 1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Zähler für S. 2		1	1	1	1	2	2	2	2	2	2	2	2	2	3	4	4
Zähler für S. 3			1	1	1	1	1	1	1	2	2	2	2	2	2	2	2
Zähler für S. 4				1	2	2	2	2	2	2	2	2	2	2	2	2	2
Seitenfehler	1	2	3	4			5		6		7	8	9	10			11

c) First-In-First-Out (FIFO):
Es wird diejenige Seite verdrängt, auf die sich am längsten im Speicher befindet.

Seitenreferenzfolge	3	1	5	8	8	1	2	4	3	4	2	5	6	7	1	1	5
Seitenrahmen 1	3	3	3	3	3	3	2	2	2	2	2	2	6	6	6	6	6
Seitenrahmen 2		1	1	1	1	1	1	4	4	4	4	4	4	7	7	7	7
Seitenrahmen 3			5	5	5	5	5	5	3	3	3	3	3	3	1	1	1
Seitenrahmen 4				8	8	8	8	8	8	8	8	5	5	5	5	5	5
Seitenfehler	1	2	3	4			5	6	7			8	9	10	11		

d) Das generelle Problem der Seitenverdrängungsmechanismen besteht darin, dass es nicht vorhersagbar ist, welche Seiten in der Zukunft noch benötigt werden und welche nicht. Zwar ist es stochastisch gesehen wahrscheinlicher, dass die zuletzt geladenen Seiten vom Prozess benötigt werden, jedoch gibt es unter Umständen auch sehr alte Seiten, auf die immer wieder zugegriffen werden muss.

In der Vorlesung wurde eine mögliche Variante eines Second-Chance-Verfahrens betrachtet. Die Idee besteht darin, dass jede Seite ein sogenanntes Zugriffsbit $z(s)$ besitzt und sich den Zeitpunkt des letzten Ladens $t(s)$ merkt. Das Zugriffsbit $z(s)$ wird beim Laden der Seite s gelöscht und $t(s)$ dementsprechend mit der aktuellen Zeit gesetzt. Bei jedem Zugriff auf die Seite s setzt man $z(s)$, wenn sich die Seite bereits im Working-Set befindet, lässt $t(s)$ dagegen unverändert.
Sollte es notwendig werden, eine Seite zu verdrängen, so bildet man eine Menge V, die alle Seiten s_i enthält, für die $z(s_i)$=gelöscht gilt. Aus dieser Menge wählt man genau die Seite $s_{verdrängen}$ aus, die den ältesten Zeitstempel $t(s_{verdrängen}) < t(s_i)$ besitzt. Zusätzlich löscht man das Zugriffsbit $z(s_k)$ für alle Seiten s_k, für die $t(s_k) < t(s_{verdrängen})$ gilt (Anmerkung: das Relationszeichen a<b bedeutet, dass a älter als b ist).
Tritt jedoch der Fall ein, dass V leer ist, so wird die älteste Seite verdrängt, was dem FIFO-Verfahren entspricht.

In einer Übersicht stellt sich der Algorithmus beim Seitenzugriff so dar:

Operation	*z(s)*	*t(s)*
Seite neu laden	löschen	= aktuelle Zeit
auf bereits geladene Seite zugreifen	setzen	unverändert

Im zeitlichen Verlauf wird nur *z(s)* geändert, *t(s)* bleibt dagegen konstant. Die Verdrängung ist etwas komplexer zu visualisieren.
Die untenstehende Tabelle arbeitet mit Indizes, um die Verwaltungsabläufe zu klären. Bei der Seitenreferenzfolge bedeutet der einzelne Index die Position innerhalb der Folge, er ist demzufolge eine fortlaufende Nummer.
Innerhalb der Seitenrahmen stellt die erste Zahl *z(s)* dar, die zweite *t(s)*. Für *z(s)* habe ich aus Platzgründen eine Kurzschreibweise benutzt, es steht 0 für gelöscht und 1 für gesetzt. Wie man sieht, wird *z(s)* in allen dunkelgrauen Felder auf 1 gesetzt. Die Situationen, in denen *z(s)* durch Verdrängungen in anderen Seitenrahmen gelöscht wird, sind blau markiert.

Seitenreferenzfolge	3_0	1_1	5_2	8_3	8_4	1_5	2_6	4_7	3_8	4_9	2_{10}	5_{11}	6_{12}	7_{13}	1_{14}	1_{15}	5_{16}
Seitenrahmen 1	$3_{0,0}$	$3_{0,0}$	$3_{0,0}$	$3_{0,0}$	$3_{0,0}$	$3_{0,0}$	$2_{0,6}$	$2_{0,6}$	$2_{0,6}$	$2_{0,6}$	$2_{1,6}$	$2_{0,6}$	$2_{0,6}$	$7_{0,13}$	$2_{0,6}$	$2_{0,6}$	$2_{0,6}$
Seitenrahmen 2		$1_{0,1}$	$1_{0,1}$	$1_{0,1}$	$1_{0,1}$	$1_{1,1}$	$1_{1,1}$	$1_{0,1}$	$3_{0,8}$	$3_{0,8}$	$3_{0,8}$	$5_{0,11}$	$5_{0,11}$	$5_{0,11}$	$5_{0,11}$	$5_{0,11}$	$5_{1,11}$
Seitenrahmen 3			$5_{0,2}$	$5_{0,2}$	$5_{0,2}$	$5_{0,2}$	$5_{0,2}$	$4_{0,7}$	$4_{0,7}$	$4_{1,7}$	$4_{1,7}$	$4_{0,7}$	$4_{0,7}$	$4_{0,7}$	$1_{0,14}$	$1_{1,14}$	$1_{1,14}$
Seitenrahmen 4				$8_{0,3}$	$8_{1,3}$	$8_{1,3}$	$8_{1,3}$	$8_{1,3}$	$8_{1,3}$	$8_{1,3}$	$8_{1,3}$	$8_{0,3}$	$6_{0,12}$	$6_{0,12}$	$6_{0,12}$	$6_{0,12}$	$6_{0,12}$
Seitenfehler	1	2	3	4			5	6	7			8	9	10	11		

Der Vorteil dieser Vorgehensweise gegenüber den anderen Verdrängungstechniken besteht darin, dass auch Seiten, die schon recht lange im Speicher sind, eine Möglichkeit haben, sich vor Verdrängung zu schützen, wenn auf sie nur entsprechend oft zugegriffen wird. Trotzdem ist das Verfahren in der Lage, alte, aber nicht mehr benutzte Seiten ordnungsgemäß zu verdrängen. Der technische Aufwand ist recht gering, deshalb wird Second-Chance vielfältig eingesetzt (z.B. in UNIX).

Die hier besprochenen Verfahren benötigen bei der gegebenen Seitenreferenzfolge genau 11 Verdrängungen. Dies lässt den naiven Schluss zu, dass kein Algorithmus eindeutige Vorteile besitzt, lediglich der technische Aufwand unterscheidet sich. Naturgemäß kann man anhand dieser kurzen und eventuell ungünstig gewählten Folge keine eindeutigen Differenzen ausmachen. Beobachtungen beim praktischen Einsatz der Verfahren erlauben aber den Schluss, dass Second-Chance die zu bevorzugende Methode ist. Lediglich bei stark eingeschränkten Hardwarevoraussetzungen sollte man die einfacher aufgebauten Techniken LRU bzw. FIFO verwenden.

Aufgabe 2

Es sind 5 Prozesse hinsichtlich ihrer Ausführungszeit auf einem Single-Prozessor-System zu optimieren:

Prozess	erwartete CPU-Belegungszeit	Priorität
1	10	3
2	1	1
3	2	3
4	1	4
5	5	2

Der Prozess, der gerade CPU-Zeit in Anspruch nimmt, ist grau hinterlegt, der Tabellenkopf spiegelt die Zeitachse wider.

Da bei „First Come, First Served“ die Prioritäten keine Rolle spielen und alle Prozesse zeitgleich eintreffen, arbeite ich sie in der Reihenfolge ihrer Prozessnummer ab:

FCFS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Prozess 1																			
Prozess 2																			
Prozess 3																			
Prozess 4																			
Prozess 5																			

Für „Shortest Job First“ sind die Prioritäten ebenfalls unwichtig, es zählt nur die vermutete CPU-Belegungszeit. Sollten zwei Prozesse eine identische Dauer benötigen, so entscheidet die niedrigere Prozessnummer:

SJF	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Prozess 1																			
Prozess 2																			
Prozess 3																			
Prozess 4																			
Prozess 5																			

Jetzt werden bei non-preemptiven Scheduling erstmals die Prioritäten verwendet, bei Gleichheit wird der Prozess mit der niedrigeren Prozessnummer ausgeführt:

non-preemptiv	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Prozess 1																			
Prozess 2																			
Prozess 3																			
Prozess 4																			
Prozess 5																			

Das preemptive „Robin Round“ arbeitet die Prozesse in aufsteigender Prozessnummernfolge für jeweils eine 1 Zeiteinheit ab:

preemptiv RR 1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Prozess 1																			
Prozess 2																			
Prozess 3																			
Prozess 4																			
Prozess 5																			

Jetzt wird die gleiche Technik angewandt, nur beträgt die Zeitscheibendauer diesmal 2 Einheiten:

preemptiv RR 2	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Prozess 1																			
Prozess 2																			
Prozess 3																			
Prozess 4																			
Prozess 5																			

Ein wichtiges Merkmal wäre für mich die durchschnittlich vergangene Zeit bis zur Erledigung eines Prozesses:

Verfahren	P1	P2	P3	P4	P5	∅	Task-Switches
FCFS	10	11	13	14	19	13,4	5
SJF	19	1	4	2	9	7,0	5
non-preemptiv	16	1	18	19	6	12,0	5
preemptiv RR 1	19	2	7	4	14	9,2	15
preemptiv RR 2	19	3	5	6	15	9,6	10

Nach diesem Kriterium wäre Shortest-Job-First die beste Strategie, da sie im Durchschnitt nur 7 Zeitscheiben zur Bearbeitung eines Prozesses benötigt.
In der Praxis ist aber selten das vollständige Beenden eines Prozesses relevant, oft will man schon zwischenzeitliche Ergebnisse haben. Als Beispiel wäre hier die Druckausgabe zu nennen, die möglichst kontinuierlich Daten zum Drucker schicken sollte. Wenn diese Peripherie deutlich langsamer als der Prozessor ist, dann wäre sicherlich eine der pre-emptiven Robin-Round-Strategien vorzuziehen. Dabei würde meine Wahl auf die zweite fallen, da sie deutlich weniger Task-Wechsel benötigt und daher weniger Betriebssystem-interventionen erfordert.

Die Verwendung der angegebenen Prioritäten basieren i.d.R. auf Round-Robin Varianten. Im Beispiel stehen die Prioritäten vor Beginn bereits fest, d.h. sie sind statisch. Moderne Betriebssysteme sind in der Lage, die Prioritäten dynamisch zu verteilen, bzw. die vorgegebenen Prioritäten leicht zu verändern, um so die subjektive Performance des Gesamtsystems zu steigern und das sogenannte Starvation einzelner niederpriorer Prozesse durch Aging hochpriorer Prozesse zu verhindern.
Zusätzlich könnte man noch zeitkritische Echtzeit-Prozessen bevorzugen, indem man mehrere Warteschlangen, die auch nach unterschiedlichen Verfahren funktionieren können, kombiniert. Eine andere Idee ist, die Priorität der benötigten Rechenzeit anzupassen, da lange laufende Prozesse oft weniger kritisch bezüglich der Bearbeitungszeit sind.

Ich werde die Aufgabenstellung derart schedulen, dass ich den Prozessen jeweils Zeitscheiben der maximalen Größe *5 minus Priorität* zuordne (z.B. ist dies 5-3=2 für Prozess 1). Grundlage dafür ist erneut Round-Robin. Die Zuteilungsreihenfolge entspricht der Prioritätsverteilung:

RR prior	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Prozess 1, Pr. 3																			
Prozess 2, Pr. 1																			
Prozess 3, Pr. 3																			
Prozess 4, Pr. 4																			
Prozess 5, Pr. 2																			

Im Gantt-Diagramm ist gut zu erkennen, dass auch Prozess 4, der die niedrigste Priorität hat, Rechenzeit zugeteilt bekam. Hätte man nicht Round-Robin, sondern Highest-Priority-First-Served verwendet, so wäre er erst in Takt 19 bearbeitet worden, d.h. 10 Takte später. Der Vorteil, dass Prozess 1 und 5 jeweils 1 Takt früher fertig gewesen wäre, fällt dagegen sehr gering aus.

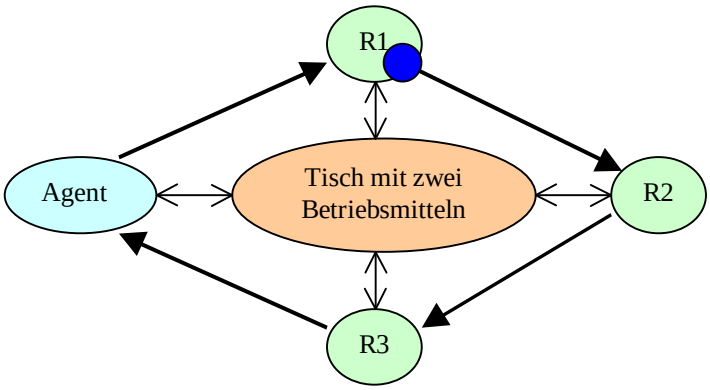
Für Echtzeitsysteme ist das Verfahren Rate-Monotonic-Scheduling interessant. Das Grundprinzip beruht darauf, dass von den bereiten Threads jeweils der mit der höchsten Priorität ausgeführt wird. Allerdings müssen dann alle Threads streng periodische CPU-Bursts vorweisen, was sich leider nicht auf die Aufgabenstellung anwenden lässt.

Aufgabe 3

Ich gehe davon aus, dass jeder Raucher und auch der Agent die gleiche Priorität haben und niemand häufiger als jemand anderes „Rechenzeit“ zugeteilt bekommt. Alle 4 Personen sitzen an einem Tisch, zusätzlich gebe ich eine Marke (in der Graphik blauer Kreis bei R1) aus.

Nur wer im Besitz dieser Marke ist, darf sich die auf dem Tisch bereitliegenden Betriebsmittel anschauen. Sollte es sich um genau die beiden ihm fehlenden handeln, so nimmt er sie, raucht in aller Ruhe und legt den nicht verbrauchten Rest wieder zurück. Danach gibt er die Marke - kooperativ wie er ist – weiter. Der Agent tauscht, wenn er an der Reihe ist, das Betriebsmittel nach dem Zufallsprinzip aus. Die Gleichstellung der Raucher bzw. des Agenten bewirkt, dass die Marke stets die gleiche Runde geht. In dieser Runde kommt jeder genau einmal zum Zuge.

Der in der folgenden Grafik veranschaulichte Rundgang der Marke im Uhrzeigersinn muss nicht unbedingt in dieser Reihenfolge erfolgen, es wäre auch R3-Agent-R1-R2 denkbar usw., jedoch darf in *einer* Runde niemand *zweimal* die Marke besitzen, d.h. alle sind gleichberechtigt. Da der Agent per Definition nicht raucht, darf er zusätzlich bereitgestelltes deutsches Mineralwasser trinken.



Die blaue Marke steht die Ressource dar, auf die jeder gerne Zugriff hätte. Ihr wird eine Semaphore zugeordnet. Die Handlungsanweisungen eines typischen Rauchers sehen dann folgendermaßen aus:

```
bis in alle Ewigkeit tue
      wait(Marke) { P-Operation }
      if (EigenesBetriebsmittel passt zu BetriebsmittelAufTisch)
            Nimm(BetriebsmittelAufTisch)
            Rauche
            LegeZurück(BetriebsmittelAufTisch)
      free(Marke) { V-Operation }
      signal(MarkeAnRechtenNachbar)
```

Der Agent führt dann die Operationen aus:

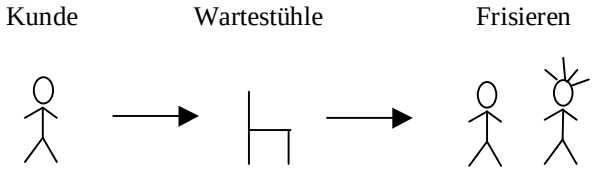
```
noch länger als bis in alle Ewigkeit tue
      wait(Marke) { P-Operation }
      Nimm(BetriebsmittelAufTisch)
      LegeAufTisch(Betriebsmittel[Zufall1], Betriebsmittel[Zufall2])
      free(Marke) { V-Operation }
      signal(MarkeAnRechtenNachbar)
```

Es ist noch zu klären, ob zu Beginn bereits Betriebsmittel auf dem Tisch liegen. Wenn nicht, wäre es vielleicht sinnvoll, die `if`-Bedingung der Raucher etwas zu verfeinern und diesen Sonderfall aufzunehmen. Ebenso dürfte der Agent dann nichts vom Tisch wegnehmen (da ja auch nichts draufliegt), höchstens einen Schluck trinken.

Als militanter Nichtraucher möchte ich noch ergänzen, dass Rauchen nicht nur die eigene Gesundheit, sondern auch die der anderen zerstört. Die entstehenden Kosten behindern den gesellschaftlichen Fortschritt in nicht unerheblicher Weise und schüren gewaltige soziale Konflikte.
Alle, die mir diese Äußerung jetzt übelnehmen, sollten sich Deuterium besorgen (schwerer Wasserstoff, den gibt es in recht hoher Konzentration im Meerwasser vor der Küste Norwegens), etwas Tritium (überschwerer Wasserstoff, der ist schon komplizierter zu beschaffen, da stark radioaktiv) und als Katalysator Lithium nehmen (das extrahiert man aus (jetzt kommt‘s !): Tabakblättern). Die kritische Masse der damit gebauten Wasserstoffbombe müsste bei 3 bis 4 kg liegen, die Reaktion bringt man am besten mit konventionellem Sprengstoff in Gang, die Russen griffen jedoch lieber auf eine Uran-basierte Nuklearbombe zurück.
Peace !

Aufgabe 4

Die typische Ablauf in einem Frisiersalon gestaltet sich derart, dass ein Kunde das Geschäft betritt und nach einem freien Wartestuhl Ausschau hält. Findet er keinen, so verlässt er sofort den Laden. Sobald ein Friseur unbeschäftigt ist, nimmt er den nächsten wartenden Kunden an die Hand, tanzt fröhlich und verschönert dessen Haarpracht.



In dieser Aufgabe sehe ich die Anzahl der Warte- und Frisierstühle als die beschränkte Ressource an, der eine Semaphore zur Verfügung gestellt wird. Ich initialisiere die Semaphore mit der Gesamtzahl der Frisierstühle im Friseursalon. Weiter gehe ich davon aus, dass man den aktuellen Status der Semaphore überprüfen kann. Die Bedingung `!IsFreierStuhl` entspricht der Tatsache, dass die Semaphore größer-gleich als 0 minus Wartestühle ist, `KeinKundeDa` steht dafür, dass die Semaphore ihren ursprünglichen Initialisierungswert angenommen hat.

Der Pseudocode eines Kunden sieht dann so aus:

```
if !IsFreierStuhl
      VerlasseSalon
if KeinKundeDa
      AnbrüllenBisWirklichWachDanachVerklagen
wait(Stuhl)
FrisiertWerden
free(Stuhl)
```

Und der Pseudocode der Friseure

```
wiederhole bis Ladenschluss
      solange !KeinKundeDa
            Frisieren
      SchnarchenFürsVaterland
```

Nicht ganz sicher bin ich mir bei der Operation `FrisiertWerden`. Sie impliziert, dass man eventuell noch auf einen Friseur warten muss. Diese Tatsache könnte man mit einer weiteren Semaphore darstellen.