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

## Aufgabe 1

a) Scheduling (frei übersetzt: Ablaufplanung) ist die Vergabe von Betriebsmitteln an Prozesse in einem Multitaskingsystem mit dem Ziel der Optimierung des Systemverhaltens.
Dabei können unterschiedliche Ziele verfolgt werden, deren Gewichtung vom gewünschten System abhängt:

- Auslastung der Betriebsmittel
- Antwortzeiten
- Durchsatz, d.h. Anzahl der pro Zeiteinheit bearbeiteten Aufträge
- Termine, Fristen
- eventuell noch Fairness, reproduzierbares Systemverhalten usw.

(frei nach: Rechenberg, Pomberger: Informatik-Handbuch, Hanser-Verlag, 2.Auflage 1999)

b) In einer Datenstruktur R (= ready) seien alle bereiten Prozesse/Threads aufgelistet. Jedes Mal, wenn das Betriebssystem die Kontrolle erhält, wird der nächste auszuführende Prozess nach folgendem Verfahren bestimmt:

```
while not ctrl-alt-del
   if !(R is empty)
         EarliestDeadline = TagDesJüngstenGerichtes
         NextProcess = null
         for every r in R do
               if Deadline(r) < EarliestDeadline
                     EarliestDeadline = Deadline(r)
                     NextProcess = r
         Execute(NextProcess)
```

Der grundlegende Unterschied zwischen pre-emptiven und nicht-preemptiven System besteht in der Regelmäßigkeit, mit der das Betriebssystem die Kontrolle erhält. In nicht-preemptiven (auch kooperativ genannten) Umgebungen liegt es allein in der Verantwortung des gerade aktiven Prozesses, wann er das Betriebssystem zum Zuge kommen lässt. Es ist damit sogar möglich, das Betriebssystem komplett zu blockieren und die Kontrolle des Systems an sich zu reißen. Dies behindert die Erfüllung anstehender Fristen und führt damit zur Systeminstabilität.
Mag dies in Echtzeitumgebungen manchmal wichtig und notwendig sein, so wird heutzutage doch eher der preemptive Fall bevorzugt. Dabei kann sogar berücksichtigt werden, dass eventuell während der Ausführung des letzten Prozesses ein neuer Prozess erzeugt wurde, der eine zeitlich nähere Deadline hat. Gegenüber der kooperativen Lösung ist man damit in der Lage, alle Fristen stets zu erfüllen, wenn dies möglich ist. Es muss noch erwähnt werden, dass jetzt auch laufende Prozesse unterbrochen werden können, was eine Absicherung der Prozessumgebung während des Task-Switches zur Folge hat, die wiederum Rechenzeit kostet. Dies ist der Preis, den man zahlen muss, um auch Amok-laufende Programme kontrollieren zu können.

c) Die Verteilung der Pakete der Prozesse sieht wie folgt aus:

| Paket | Ankunftszeit Prozess 1 | Deadline Prozess 1 | Ankunftszeit Prozess 2 | Deadline Prozess 2 |
|---|---|---|---|---|
| 1 | 0 | 2 | 0 | 5 |
| 2 | 2 | 4 | 6 | 10 |
| 3 | 4 | 5 | - | - |
| 4 | 6 | 8 | - | - |
| 5 | 8 | 10 | - | - |

Im folgenden sei die Ausführungszeit eines Prozesses dunkelgrau markiert, die Wartezeit hellgrau. Das Verletzen einer Frist ist rot gekennzeichnet. Die Zahlen in den Balken stehen für die Paketnummer.
Earliest Deadline First:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Prozess 1 | 1 | | 2 | | 3 | | 4 | | 5 | |
| Prozess 2 | 1 | | | | | | 2 | | | |
| Zeitachse | 1 | | 3 | | 5 | | 7 | | 9 | |

Es ist zu erkennen, dass der Scheduling-Mechanismus zweimal fehlschlägt und Fristen verletzt (Zeitscheiben 4 und 9). In Zeitscheibe 5 ist das System hingegen idle.

Der Algorithmus Rate-Monotonic arbeitet leider auch nicht effizienter:

| Prozess 1 | 1 | | 2 | | 3 | | 4 | | 5 | |
|---|---|---|---|---|---|---|---|---|---|---|
| Prozess 2 | 1 | | | | | | 2 | | | |
| Zeitachse | 1 | | 3 | | 5 | | 7 | | 9 | |

Die Problematik ist die fast gleiche, diesmal sind aber nur die beiden Pakete von Prozess 2 betroffen, alle Pakete von Prozess 1 wurden fristgerecht bearbeitet. Erneut kann Zeitscheibe 5 nicht ausgenutzt werden.

In der Theorie benötigt Prozess 1 für seine 5 Pakete insgesamt 5 Zeitscheiben. Aufgrund der zeitlichen Einschränkung der Eingriffsmöglichkeiten des Betriebssystems verbraucht ein Paket von Prozess 2 effektiv 3 Zeitscheiben. Da 2 Pakete eintreffen, muss der Prozess 6 Zeitscheiben zur Verfügung gestellt bekommen. In der Summe sind 11 Zeitscheiben notwendig. Dies ist unter Einhaltung der Fristen mit keinem Verfahren möglich.

## Aufgabe 2

a) *nicht bearbeitet*

b) Die einfacher zu implementierende Variante der nachrichtenbasierten Prozessinteraktion ist die Verwendung von synchronen Meldungen, da der Empfänger nicht puffern kann/muss. Allerdings muss dazu sichergestellt werden, dass sowohl Sender als auch Empfänger bei der Übertragung der Meldung bereit sind. Der Sender benötigt zusätzlich noch eine Rückmeldung vom Empfänger und bleibt bis zu ihrem Erhalt im blockierten Zustand. Dies stellt sicher, dass alle versandten Nachrichten auch vom Empfänger bearbeitet wurden. Nachteilig daran ist, dass der Sender in seiner Leistungsfähigkeit durch den Empfänger begrenzt wird. Besonders wenn ein Sender mehrere Empfänger bedienen muss oder nur unidirektionale Verbindungen möglich sind (z.B. Multimedia-Streamingdienste wie digitales Fernsehen oder Internetmusik via RealAudio), ist die asynchrone Kommunikation die günstigere. Der Sender wird nie blockiert, allerdings kann er auch nicht sicherstellen, ob alle verschickten Daten auch ankamen bzw. verarbeitet wurden. An dieser Stelle sollte das Datenformat Informationsverluste überbrücken können (durch Interpolation etc.). Außerdem muss der Empfänger einen Empfangspuffer für bereits angekommene, aber noch nicht von ihm bearbeitete Meldungen bereitstellen.

## Aufgabe 3

Asynchrone Meldungen blockieren den Sender generell nicht und der Empfänger wird nur dann blockiert, wenn keine Meldungen in seinem Puffer bereitliegen. Diese Technik erlaubt die größtmögliche Ausnutzung der Ressourcen, da weder unnötige Prozessblockierungen noch zusätzliche Übertragungsdaten generiert werden. Ideal eignet sich dieses Verfahren für verlusttolerantes Streaming, im Internet i.d.R. auf Basis von UDP.

Synchrone Meldungen erfordern, dass der Sender nach Abschicken der Meldung solange blockiert, bis er vom Empfänger eine Bestätigung erhalten hat. Der Sender wird in seiner Kapazitätsausnutzung begrenzt, der Empfänger kann aber einfacher aufgebaut werden als bei asynchronen Meldungen. Im Internet braucht man diese Zuverlässigkeitseigenschaften und baut TCP/IP dementsprechend auf, da man eine Übertragungsgarantie von Daten wünscht.

Asynchrone Aufträge spielen ihre Vorteile der Pufferung und Blockade-Vermeidung wieder bei mehreren Empfängern aus. Diese können unterschiedlich schnell sein, ein typisches Beispiel sind Mail-Server, die Benutzer mit unterschiedlichen Bandbreiten und Latenzzeiten verkraften müssen. Das eindeutige Identifizierungsmerkmal beim Kommunikationsdialog ist das Paar IP-Adresse/Portnummer.

Synchrone Aufträge sind einfacher zu implementieren, da, wie bei synchronen Meldungen, der Sender solange blockiert bleibt, bis er das Ergebnis vom Empfänger erhalten hat. Die Ressourcenausnutzung bleibt daher stark unter ihrer theoretischen Kapazität. Der Handshake beim Aufbau einer passwortgeschützten Verbindung zu einem Internetanbieter via Modem blockiert den Sender (d.h. den PC, der ins Internet möchte) solange, bis der Remote-Computer das Passwort überprüft hat und die Datenleitung freigibt.

RPCs (Remote Procedure Calls) bauen auf der synchronen Auftragskommunikation auf. Sie erlauben den Aufruf von Prozeduren über Adressgrenzen hinweg, meist sogar auf anderen Rechnern. Das dafür erforderliche Late-Binding dynamisch zur Laufzeit ist nicht ganz unproblematisch und muss von den unterstützenden Bibliotheken abgesichert werden. Ein plattformübergreifendes Beispiel ist RMI für Java, in meinen Augen leistungsfähiger ist das nur für Windows-Betriebssysteme verfügbare DCOM (Distributed Component Object Model).

Signale werden dann eingesetzt, wenn die Anzahl der zu übertragenden Daten sehr klein ist. Sie stellen eine Art Software-Interrupt dar, der die Ausführung des aktuellen Prozesses unterbricht und eine Signal-behandlungsroutine aufruft. Diese muss nach ihrem Ende ein SignalReturn() aufrufen, um den unterbrochenen Prozess weiterzuführen. Ein mögliches Signal stellen die PAINT-Nachrichten in graphischen Oberflächen dar, die das jeweilige Programm veranlassen, das eigene Fenster zu zeichnen, da es aufgrund von Verschiebungen, Vergrößerungen usw. verändert wurde.

## Aufgabe 4 (Extra-Aufgabe 1)

a)

| Prozess | Ankunftszeit | Burstzeit | Anfangspriorität |
|---|---|---|---|
| 1 | 0 | 77 | 10 |
| 2 | 3 | 30 | 7 |
| 3 | 22 | 49 | 9 |
| 4 | 110 | 8 | 10 |

Im Gantt-Diagramm stellt sich die Verteilung der Ausführungszeit durch den Scheduler wie folgt dar (Bursts sind dunkelgrau, Idle-Zeiten hellgrau, die Zahlen in den Balken sind die Prioritäten während der Zeitscheibe mit der aktuellen Position in der Prozesswarteliste als Index):

| Zeit | 0 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 | 110 | 120 | 130 | 140 | 150 | 160 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $P_1$ | $10_1$ | $9_1$ | $8_1$ | $7_3$ | $7_3$ | $7_2$ | $7_1$ | $6_3$ | $6_2$ | $6_1$ | $5_3$ | $5_3$ | $5_2$ | $5_1$ | $4_2$ | $4_1$ | $3_1$ |
| $P_2$ | 7 | $7_2$ | $7_2$ | $7_2$ | $7_2$ | $7_1$ | $6_3$ | $6_2$ | $6_1$ | $5_3$ | $5_2$ | $5_2$ | $5_1$ | | | | |
| $P_3$ | | | 9 | $9_1$ | $8_1$ | $7_3$ | $7_2$ | $7_1$ | $6_3$ | $6_2$ | $6_1$ | $5_4$ | $5_3$ | $5_2$ | $5_1$ | | |
| $P_4$ | | | | | | | | | | | | $10_1$ | | | | | |

In den Zeitscheiben 40 bis 100 ist schön zu beobachten, dass drei Prozesse mit gleicher Priorität sich um die CPU streiten, da ein Zyklus $P_3$-$P_2$-$P_1$ entsteht, der erst von $P_4$ unterbrochen wird. In einem realen System ist darauf zu achten, dass die Priorität mindestens ein Zehntel der Burstzeit sein muss, da sonst negative Werte entstehen könnten (bzw. das Betriebssystem belässt die Priorität bei min. 1). Außerdem wäre es sinnvoll, über Dis-Aging-Strategien nachzudenken, d.h. die Prioritäten nach zu langen Idle-Zeiten wieder zu inkrementieren, um so Starvation-Effekte zu vermeiden.

b)

| Zeit | 0 | 13 | 26 | 39 | 52 | 65 | 78 | 91 | 104 | 117 | 130 | 143 | 156 | 169 | 182 | 195 | 208 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $P_1$ | $10_1$ | $9_1$ | $8_2$ | $8_1$ | $7_3$ | $7_2$ | $7_1$ | $6_3$ | $6_2$ | $6_2$ | $6_1$ | $5_3$ | $5_2$ | $5_1$ | $4_2$ | $4_1$ | $3_1$ |
| $P_2$ | 7 | $7_2$ | $7_3$ | $7_3$ | $7_2$ | $7_1$ | $6_3$ | $6_2$ | $6_1$ | $5_4$ | $5_3$ | $5_2$ | $5_1$ | | | | |
| $P_3$ | | 9 | $9_1$ | $8_2$ | $8_1$ | $7_3$ | $7_2$ | $7_1$ | $6_3$ | $6_3$ | $6_2$ | $6_1$ | $5_3$ | $5_2$ | $5_1$ | | |
| $P_4$ | | | | | | | | | 10 | $10_1$ | | | | | | | |

In der Graphik habe ich die 3 ms der Aktivität des Betriebssystems pro Zeitscheibe nicht extra markiert. Ansonsten gelten die gleichen Aussagen wie in Aufgabenteil a).

## Aufgabe 5 (Extra-Aufgabe 2)

Die Semaphore dient der Überquerung der Brücke, ich nenne sie daher $s_b$. Zusätzlich muss ich fordern, dass die Semaphore nicht nur 1er Schritten in Anspruch genommen werden kann. Sollte dies nicht umsetzbar sein, so muss der Programmierer aus der V-Operation `wait(`$s_b$`,3)` eben `wait(`$s_b$`); wait(`$s_b$`); wait(`$s_b$`);` machen. Dementsprechend ist mit der P-Operation `free` zu verfahren. Diese Forderung stelle ich deshalb auf, weil ein Nashorn 15 Eingeborenen entsprechen soll, d.h. die Brücke in ihrer Kapazität voll auslastet. Somit kann ich verhindern, dass Eingeborene und Nashörner gleichzeitig auf der Brücke sind.

Neben der Semaphore verwende ich noch eine `signal`-Operation. Sie ordnet die Warteschlange der Semaphore derart um, dass alle wartenden Dorfbewohner erst *nach* dem letzten wartenden Nashorn zum Zuge kommen.
Die Initialisierung der Semaphore ist recht einfach:

```
init(sb,15)
```

Jedes Nashorn bewirbt sich um die Brücke, sperrt aber gleichzeitig den Zutritt für alle wartenden Dorfbewohner. Sollten mehrere Nashörner warten, so regelt $s_b$ die Reihenfolge durch FIFO.

```
wait(sb,15)
signal(NashornVorlassen)
BrückeÜberqueren
free(sb,15)
```

Die Eingeborenen verfahren ganz ähnlich:

```
wait(sb,1)
BrückeÜberqueren
free(sb,1)
```

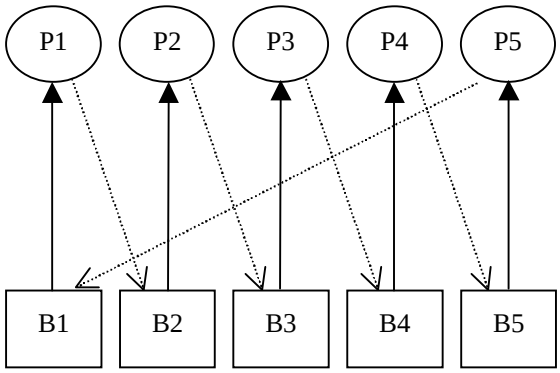
Das Signal `NashornVorlassen` sieht so aus:

```
find position i of least recent Nashorn in WaitingProcessList
if i = null
      insert Nashorn at 0
else
      insert new Nashorn at i+1
SignalReturn
```

Dabei ist besonderer Wert darauf zu legen, dass sich kein Nashorn vor ein anderes drängelt, was auch der sportlichen Fairness entspricht. Den Eingeborenen bleibt nichts anderes übrig, als sich mit dem olympischen Gedanken – "Dabei sein ist alles" – zu motivieren.

**Aufgabe 6 (Extra-Aufgabe 3)**

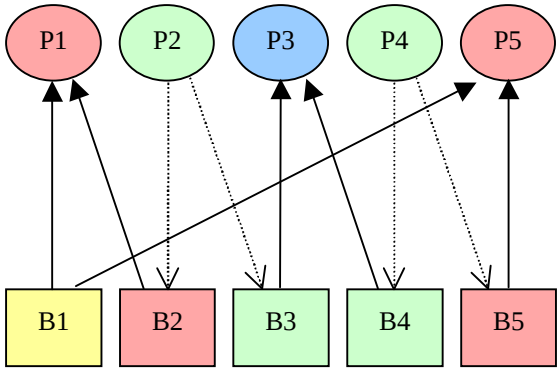
a) In der Graphik ist die Verklemmung als Zyklus zu erkennen:




Der Buchstabe P steht für Philosoph, der Buchstabe B für Betriebsmittel, was in diesem Fall die Gabeln sind. Philosoph PX hat zu seiner linken Seite Betriebsmittel BX, zu seiner rechten B(X+1 modulo 6). Die gestrichelten Pfeile sind Anforderungen, die durchgezogenen Pfeile hingegen Belegungen.
Es ist ersichtlich, dass alle Betriebsmittel B1 bis B5 durch die Philosophen belegt sind und jede weitere Anforderung nicht erfüllt werden kann. Da aber auch kein Philosoph essen kann und er erst *nach* dem Essen wieder seine Betriebsmittel freigeben würde, ist ein Deadlock entstanden.

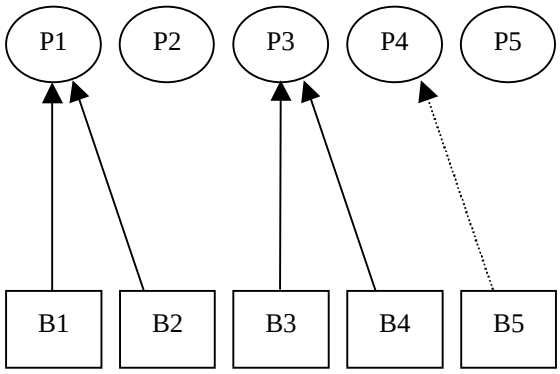
b) Es gibt grundsätzlich zwei verschiedene morphologische Typen von Philosophen: Dicke und Dünne. Dass es nicht zwischendrin gibt – Normalos – liegt in den Spagetti-Gewohnheiten begründet. Dieses Problem war

schon im alten Griechenland bekannt, obwohl die nicht einmal das Spagetti-Rezept kannten, und zieht sich seither durch alle zeitgeschichtlichen Epochen.
Die Dicken sind deshalb dick, weil sie lange und ausgiebig essen, die Dünnen müssen froh sein, dass sie ab und zu eine Nudel erwischen und sich so wenigstens das Überleben sichern können. Die beiden Philosophen, die das Pech haben, direkt neben einen Dicken zu sitzen, können solange nicht essen, wie der Dicke seinen Körperfettgehalt oral erhöht. Die anderen beiden, die das Glück hatten, nicht direkt neben dem Dicken zu sitzen, können abwechseln essen, was für die Gesundheit aber noch gefährlicher ist, weil es nicht nur dick, sondern fett macht und den Cholesterinspiegel in krankenkassengefährdende Bereiche treibt. Der Betriebsmittelgraph zeigt den Dicken (blau), die beiden Dünnen (grün) und die beiden Fetten (rot):



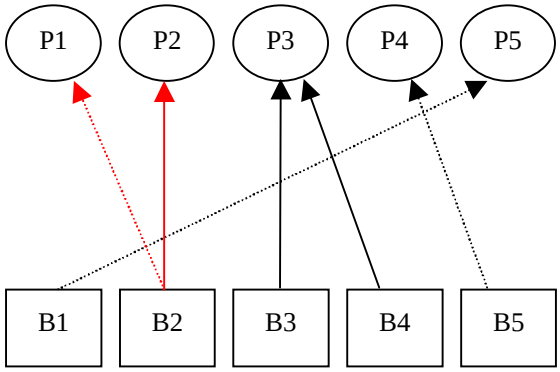

Das gelbe Betriebsmittel ist shared, d.h. wird von P1 und P5 abwechseln benutzt. Natürlich muss auch P3 mal eine Pause machen. Entweder P2 *oder* P4 kann nun essen. Da sie als Dünne fast vergessen haben, was Spagettis sind, sind sie sehr schnell wieder fertig. Diese Chance nutzt P3 und ergreift wieder beiden Gabeln. Sollten die Auszeiten von P3 in einem festen Verhältnis zu P1/P5 stehen, kann es passieren, dass ein Philosoph (entweder P2 oder P4) nie Spagettis isst.

c) Da insgesamt 5 Gabeln vorhanden sind, ist es möglich, dass stets 2 Philosophen essen. Grundvoraussetzung ist wieder, dass ein Philosoph nur dann eine Gabel greift, wenn er auch sofort die andere greift. Ist diese nicht verfügbar (da in Benutzung), so nimmt er die erste Gabel auf gar keinen Fall auf.
Zu Beginn können, rein zufällig, zwei Philosophen mit dem Essen beginnen. Die drei, die nicht essen können, haben folgende Probleme: einem fehlt die linke Gabel, einem fehlt die rechte Gabel, einem fehlen beide Gabeln.




In der Graphik essen P1 und P3, hingegen müssen P2, P4 und P5 warten. Die Idee ist nun, dass ein Philosoph, der gerade mit Essen fertig wurde, dafür sorgt, dass sein rechter Nachbar, d.h. der mit der nächsthöheren Nummer als nächster isst.
Dies erreiche ich, indem P4, dem nur die linke Gabel fehlt, sich bereits den Besitz der rechten Gabel B5 sichert (der gestrichelte Pfeil). Sobald P3 fertig ist, wird auch die linke Gabel B4 frei, so dass P4 dann essen kann. Gleichzeitig wird auch B3 frei, die sich P2 sichert.
Ein Problem tritt auf, wenn erst P1 und danach P3 fertig wird: es könnte sich P5 nun die freie rechte Gabel B1 sichern. Danach sichert sich eventuell P1 seine freie rechte Gabel B2 und blockiert damit P2s linke Gabel (die ja ebenfalls B2 ist), was in der nachfolgenden Graphik durch einen gestrichelten roten Pfeil verdeutlicht wird. Deshalb muss P1 beim Beenden des Essens an P2 ein Signal senden, dass die sofortige Aufnahme von B2 veranlasst (roter durchgezogener Pfeil).

Sobald P3 fertig ist, können sowohl P2 als auch P4 mit dem Essen beginnen. Es wird nun stets reihum gegessen. Leider werden die zur Verfügung stehenden Gabeln nicht optimal ausgenutzt, in der eben beschriebenen kritischen Situation wäre es möglich, dass P5 isst.




Der Pseudocode dafür:

```
repeat
   if BX and B(X+1 modulo 6) are free
         wait(BX and B(X+1 modulo 6))
   else
         wait(B(X+1 modulo 6))   // ergreife rechte Gabel
         if not already owning BX
               wait(BX)          // ergreife linke Gabel

   Essen

   free(BX)                      // rechte Gabel freigeben
   signal(B(X+1 modulo 6), NimmBX)
                                 // sofort an rechten Nachbarn melden,
                                 // dessen linke Gabel frei ist
   free(B(X+1 modulo 6))         // linke Gabel freigeben
until false;
```

Die blau markierte Operation wird i.d.R. nur in der Initialisierungsphase ausgeführt und muss atomar realisiert werden. Sie müsste eigentlich eher `take` heißen, da der Prozess nicht warten braucht, sondern sofort die Ressource beanspruchen kann.