Anleitung zur Selbstkontrolle bei Interaktionsdiagrammen

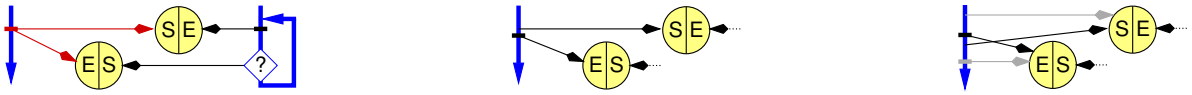
1. Stimmt die Anzahl und Art der Prozesse?
   Reine Clients mit offenem Ende, Server und anderes in Dauerschleife.

2. Stimmt die Anzahl der Kanäle?

3. Stimmt die Verdrahtung?
   Rufen alle Prozesse an den richtigen Kanälen die richtige Operation auf?

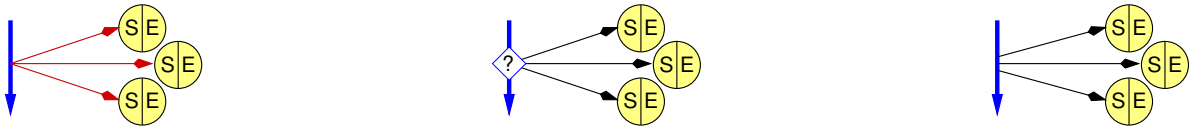
4. Stimmt die Reihenfolge der Aufrufe in jedem Prozess?

Falsch: Senden und Empfangen gleichzeitig



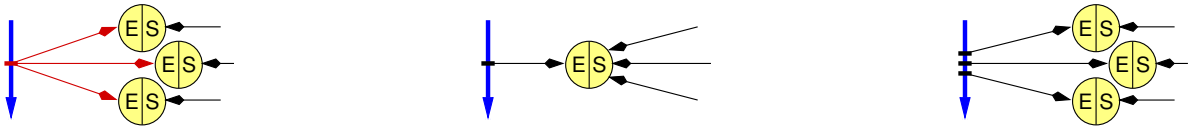
Richtig: erst Senden, dann Empfangen
seltener: erst Empfangen, dann Senden

Falsch: Senden an mehrere Kanäle gleichzeitig



Richtig: Senden an einen von mehreren Kanälen
seltener: an mehrere Kanäle nacheinander

Falsch: Empfangen an mehreren Kanälen gleichzeitig



Richtig: Empfangen an genau einem Kanal
seltener: an mehreren Kanälen nacheinander

Falsch: Antwort empfangen bevor Auftrag gesendet
Die Richtung des Befehlsstroms (blauer Pfeil) bestimmt die Reihenfolge!

Richtig: Auftrag senden, dann Antwort empfangen

Falsch: Auftragskanal als Antwortkanal mitverwenden

Richtig: Verschiedene Kanäle für jeden Zweck

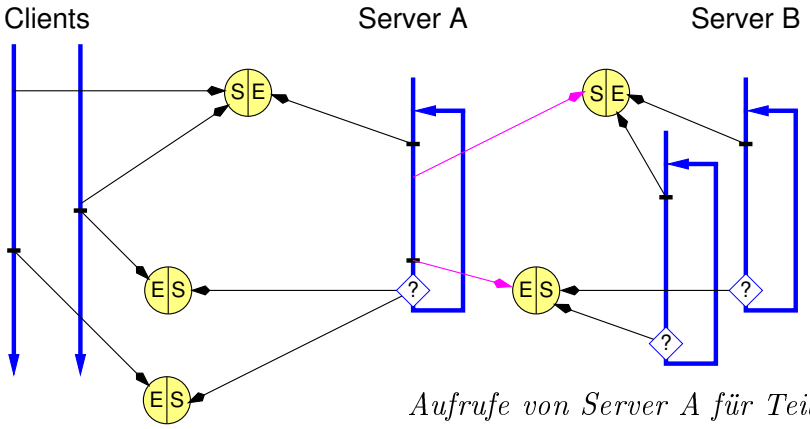
Aufgabe K1: Interaktionsmuster Client/Server

Erstellen Sie Interaktionsdiagramme nach den folgenden Beschreibungen.
Jede Teilaufgabe erweitert die vorhergehende.

a) Zwei Clients beauftragen einen Server A. Ein Client schickt seinen Auftrag asynchron, der andere synchron.

b) Server A schickt einen asynchronen Auftrag an einen Server B. Sowohl Senden als auch Empfangen erfolgen in der Arbeitsphase von Server A.

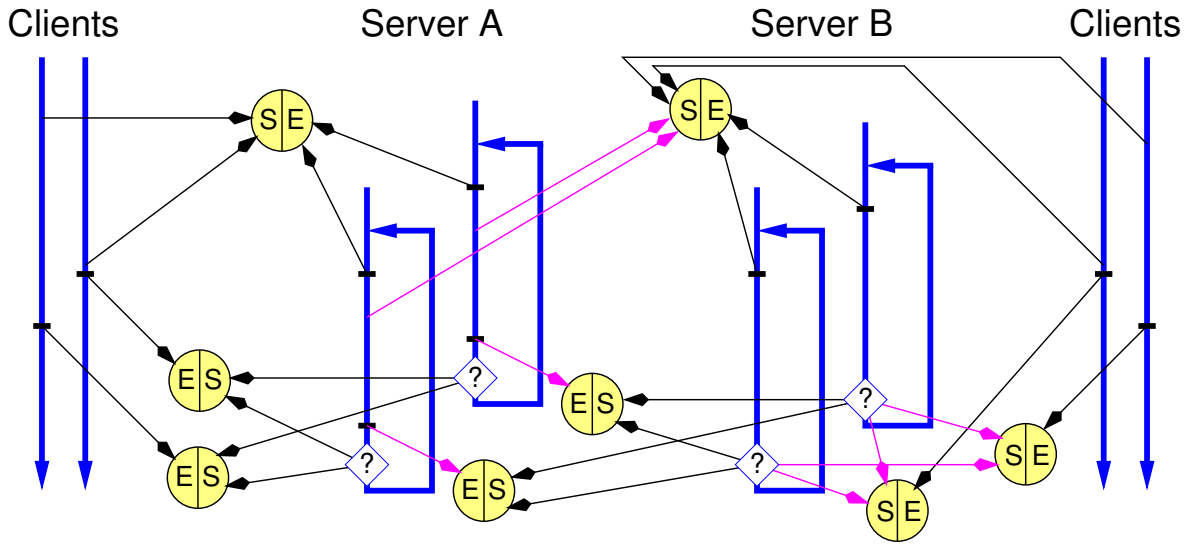
c) Server B bearbeitet Aufträge mit zwei parallelen Prozessen. (Reproduktion)



Aufrufe von Server A für Teil b in Pink.

d) Zwei weitere Clients beauftragen den Server B. Ein Client schickt seinen Auftrag synchron, der andere asynchron. *Tipp: Diese gegenüberliegend zu Teil a zeichnen.*

e) Server A bearbeitet Aufträge ebenfalls mit zwei reproduzierten Prozessen.

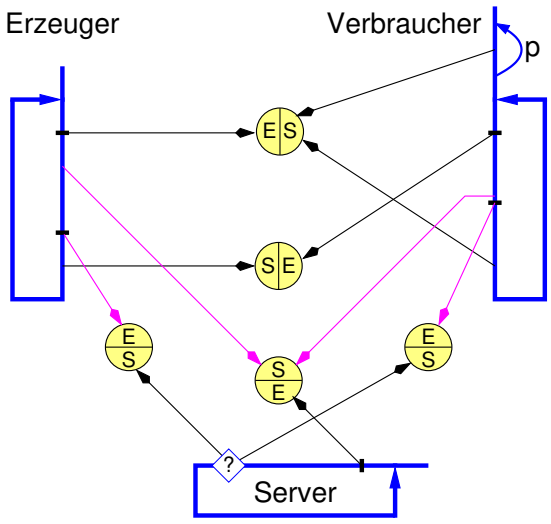


Antworten von Server B für Teil d ebenfalls in Pink.

Aufgabe K2: Interaktionsmuster Erzeuger/Verbraucher

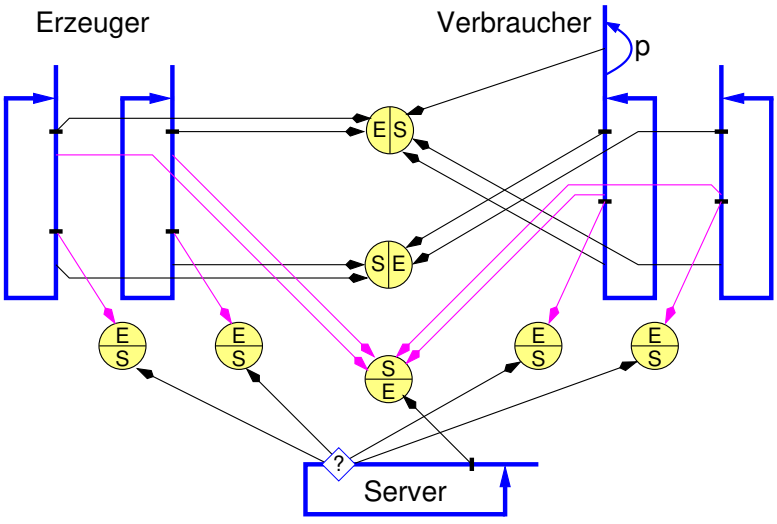
Erstellen Sie Interaktionsdiagramme nach den folgenden Beschreibungen.

a) In einem Erzeuger–Verbraucher–System zirkulieren p Nachrichten. Die Initialisierung übernimmt der Verbraucher. Sowohl Erzeuger als auch Verbraucher beauftragen in ihrer Arbeitsphase je einmal einen gemeinsamen Server. Der Erzeuger schickt seinen Auftrag asynchron, der Verbraucher nicht.



Aufrufe für die Beauftragung in Pink, zur besseren Unterscheidung.
Die Initialisierung ist hier als Schleife statt mit ... dargestellt.

b) Erzeuger und Verbraucher aus Teilaufgabe a sind reproduziert.

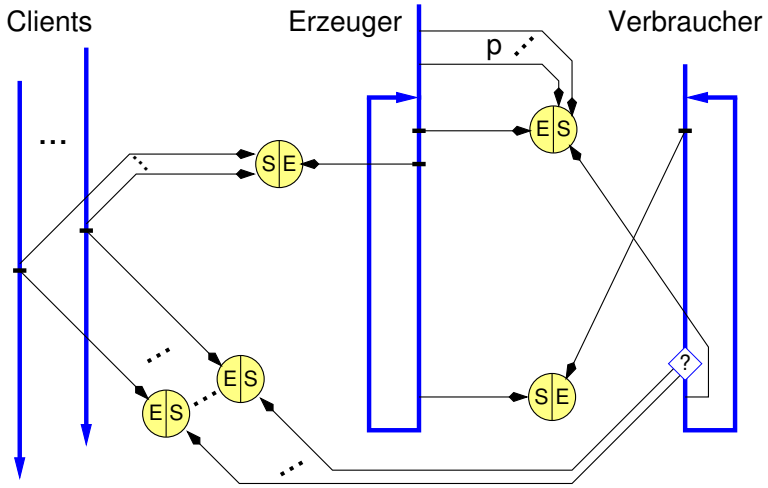


Jeder Erzeuger und Verbraucher braucht einen eigenen Antwortkanal.
Das E/V–System wird nur von einem der Verbraucher initialisiert.

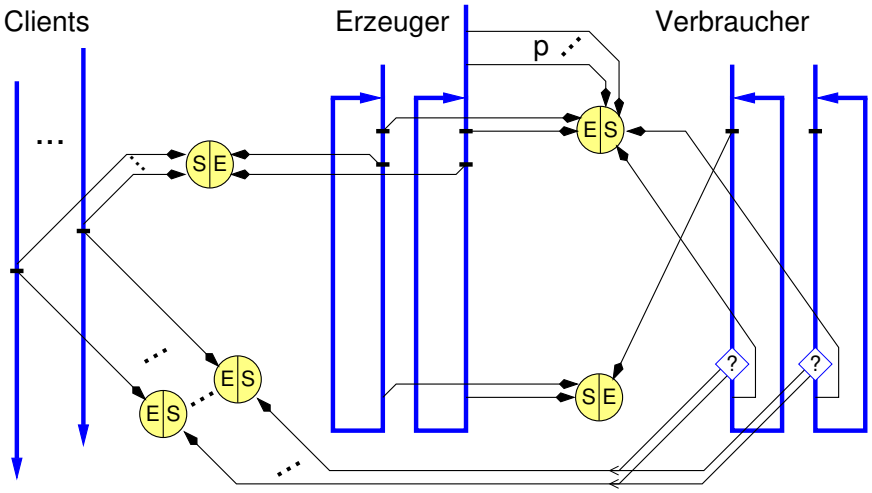
c) Welches Problem gibt es im Interaktionsmuster aus Teilaufgabe b?
Was könnte man dagegen tun? *Ohne Zeichnung!*

Erzeuger und Verbraucher bremsen sich über den gemeinsamen Server aus, der Aufträge nur nacheinander bearbeitet. Man könnte auch den Server reproduzieren. *Die Zeichnung würde allerdings zu unübersichtlich.*

d) Mehrere Clients schicken jeweils einen Auftrag und erwarten dann die Antwort. Anstelle eines Servers bearbeitet ein Erzeuger–Verbraucher–System die Aufträge. Der Erzeuger empfängt sie und beginnt mit der Bearbeitung. Der Verbraucher schließt die Bearbeitung ab und schickt die Antworten. Im Erzeuger–Verbraucher–System zirkulieren p Nachrichten, die Initialisierung übernimmt der Erzeuger.



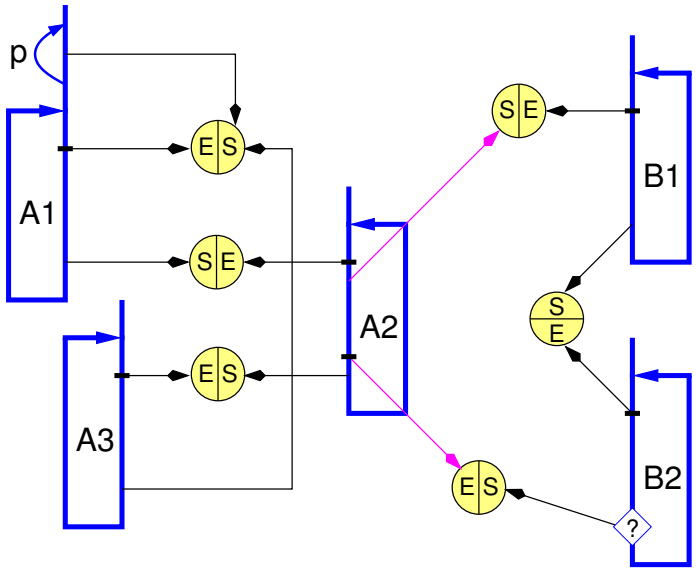
e) Erzeuger und Verbraucher aus Teilaufgabe d sind reproduziert.



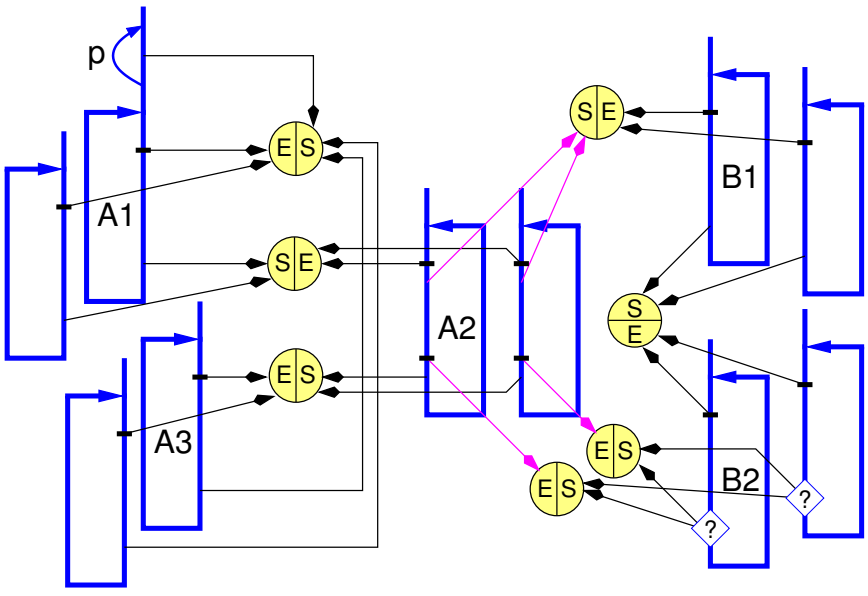
Aufgabe K3: Interaktionsmuster Fließband

Erstellen Sie Interaktionsdiagramme nach den folgenden Beschreibungen.

a) In einem dreistufigen, geschlossenen Fließband A zirkulieren p Nachrichten. Die erste Stufe übernimmt die Initialisierung. Die zweite Stufe schickt pro Durchgang einen asynchronen Auftrag und empfängt die Antwort. Diese Aufträge bearbeitet ein zweistufiges, offenes Fließband B.



b) Jeder Prozess aus Teilaufgabe a ist reproduziert.



In Stufe A1 übernimmt nur einer der Prozesse die Initialisierung.
In Stufe A2 braucht jeder Prozess einen eigenen Antwortkanal.

c) Der implementierte Algorithmus erlaubt es, das Empfangen der Antworten in die dritte Stufe zu verschieben. Die Antworten müssen weiterhin zum Auftrag passen. Ist das technisch möglich? Falls ja, wie? Falls nein, warum nicht?

Ob die Antworten passend zu den Aufträgen empfangen werden können, hängt vom tatsächlich verwendeten Kommunikationsmechanismus ab. Bei einem Message–Queueing–System zum Beispiel ist es kein Problem, auf eine bestimmte Antwort zu warten.

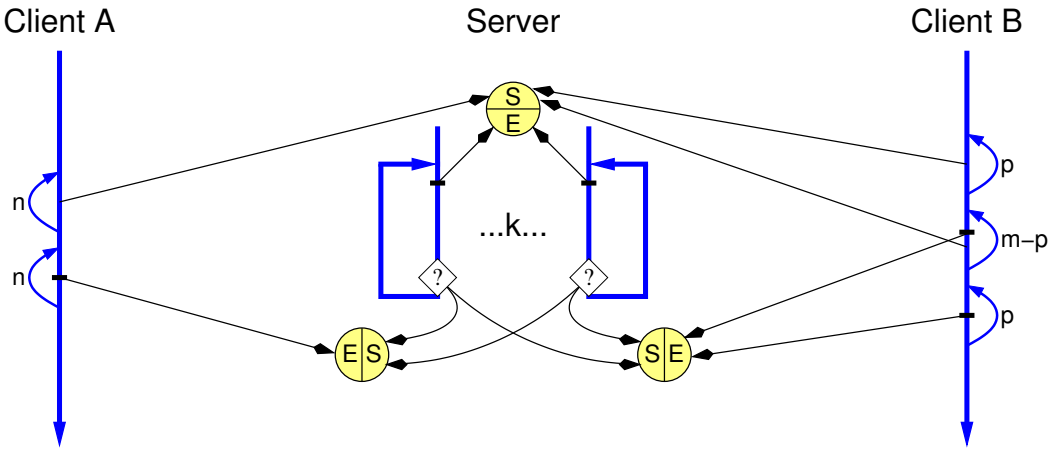
Für eine Implementierung mit den in Interaktionsdiagrammen angenommenen, sehr einfachen Kanälen genügen die Antwortkanäle pro Prozess aus Teilaufgabe b nicht. Wenn erst in Stufe A3 empfangen wird, könnte der gleiche Prozess aus Stufe A2 bereits weitere Aufträg gesendet haben. Die Antworten dieser Aufträge können in beliebiger Reihenfolge ankommen, denn das reproduzierte Fließband B ist nicht reihenfolgetreu. Aber auch ohne Reproduktion sollte man besser nicht davon ausgehen, dass die Übertragung der Nachrichten und die Bearbeitung der Aufträge reihenfolgetreu erfolgt.

Eine Lösung des Problems ist, dass die Stufe A2 für jeden Auftrag einen eigenen Antwortkanal erzeugt, der nach dem Empfangen der Antwort in Stufe A3 wieder gelöscht wird. Alternativ könnte das geschlossene Fließband auch p Antwortkanäle verwenden, die den zirkulierenden Nachrichten fest zugeordnet sind. Solche Zusammenhänge lassen sich aber nicht mehr als Interaktionsdiagramm darstellen.

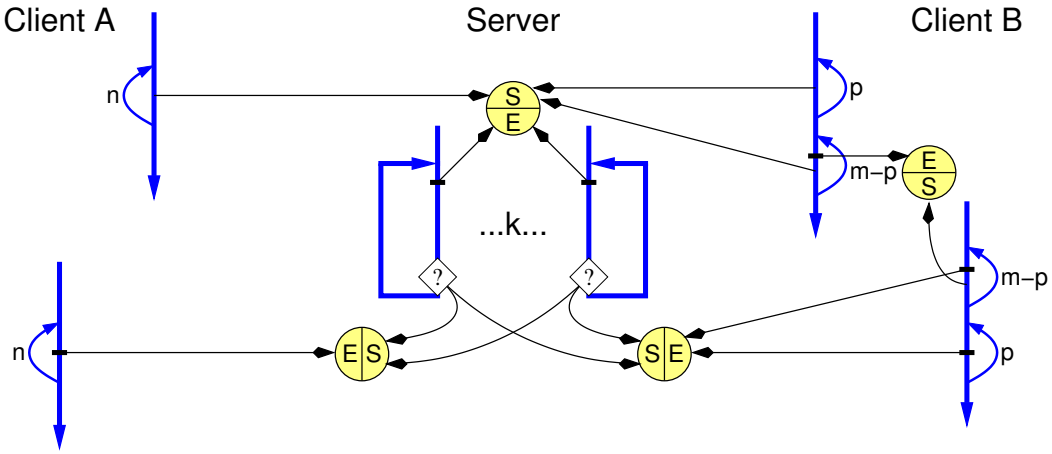
Aufgabe K4: Interaktionsmuster Pufferung

Erstellen Sie Interaktionsdiagramme nach den folgenden Beschreibungen.

a) Ein Client schickt mit unbeschränkter Pufferung n Aufträge an einen Server, der mit k Prozessen reproduziert ist. Ein anderer Client schickt m Aufträge, gepuffert mit Limit p, an den gleichen Server.



b) Beide Clients aus der ersten Teilaufgabe werden aufgeteilt, jeweils in zwei Prozesse. Der erste Prozess übernimmt das Erstellen und Senden der Aufträge, der zweite das Empfangen und Verarbeiten der Antworten. Der Server wird zur mittleren Stufe in der Kette. Der zweite Client puffert weiterhin mit Limit p.



c) Wie unterscheiden sich die entstandenen Interaktionsmuster von Fließbändern?

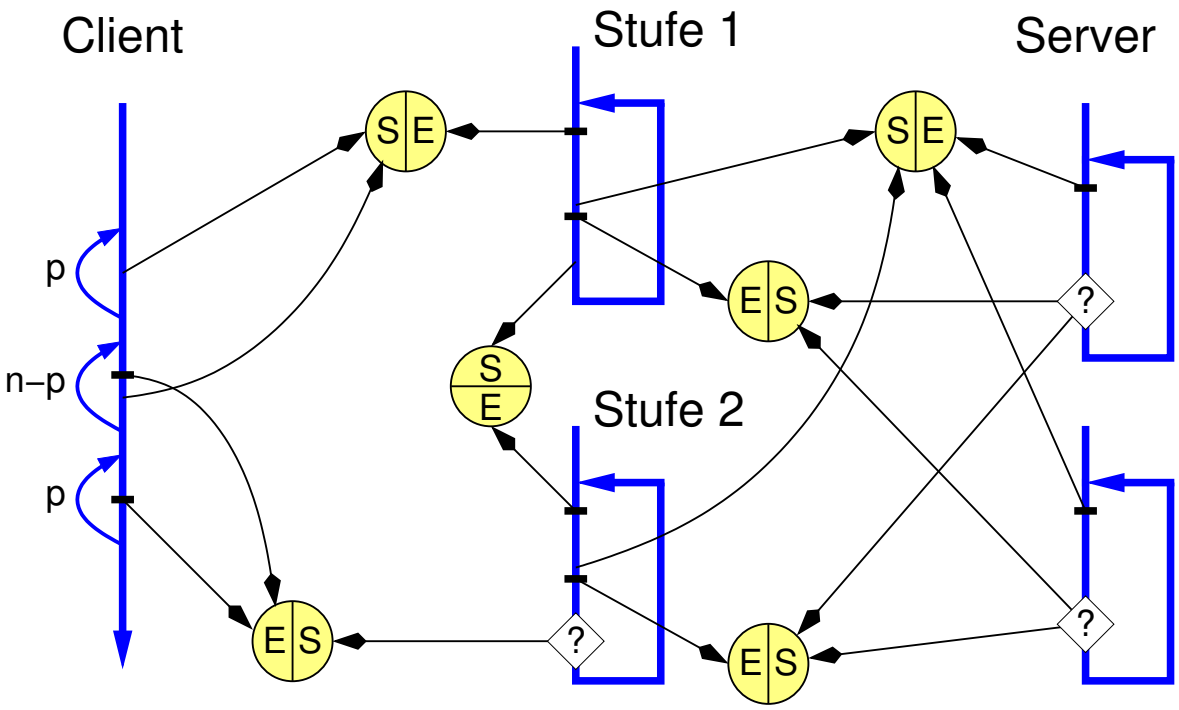
Der erste Client wird durch das Aufteilen zu einem offenen Fließband, der zweite zu einem geschlossenen. Allerdings sind die Stufen hier keine dauernd laufenden Prozesse, sondern enden nach n bzw. m Durchgängen. Die Initialisierung des geschlossenen Fließbands erfolgt nicht durch „leere“ Nachrichten an die erste Stufe, sondern durch Aufträge an die zweite. Dementsprechend meldet die letzte Stufe die letzten p Antworten auch nicht mehr an die erste Stufe zurück.

Aufgabe K5: Interaktionsdiagramm

8 Pkt

Erstellen Sie ein Interaktionsdiagramm nach der folgenden Beschreibung:

Ein Client sendet n Aufträge mit p-facher Pufferung und empfängt die Antworten. Die Aufträge bearbeitet ein zweistufiges, offenes Fließband. Die Stufen vergeben während der Bearbeitung je einen Unterauftrag an einen weiteren Server und empfangen dessen Antwort. Beide Stufen senden ihre Unteraufträge an den gleichen Server. Dieser Server ist mit insgesamt zwei Prozessen reproduziert.

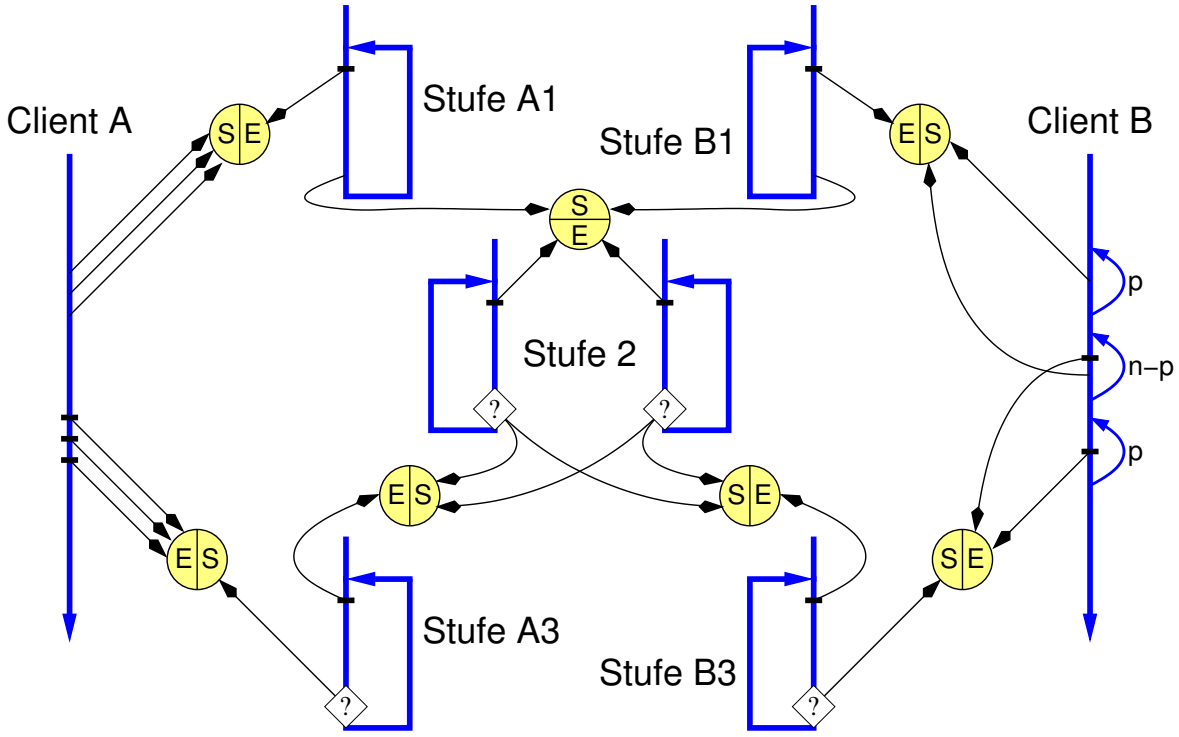


Aufgabe K6: Interaktionsdiagramm

8 Pkt

Erstellen Sie ein Interaktionsdiagramm nach der folgenden Beschreibung:

Ein Client A schickt drei Aufträge asynchron an ein dreistufiges Fließband A. Anschließend arbeitet er noch eine Weile und empfängt dann die Antworten. Ein Client B schickt n Aufträge gepuffert mit Limit p an ein dreistufiges Fließband B. Beide Fließbänder verwenden eine gemeinsame zweite Stufe, die mit insgesamt zwei Prozessen reproduziert ist.

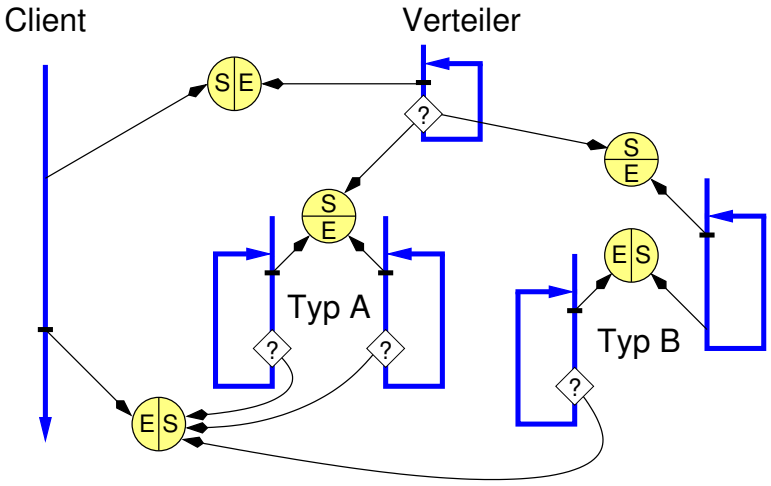


Punkte: asyn. Aufträge 2, Pufferung 2, Fließbänder 2, gemeinsame Stufe 1, Repro 1

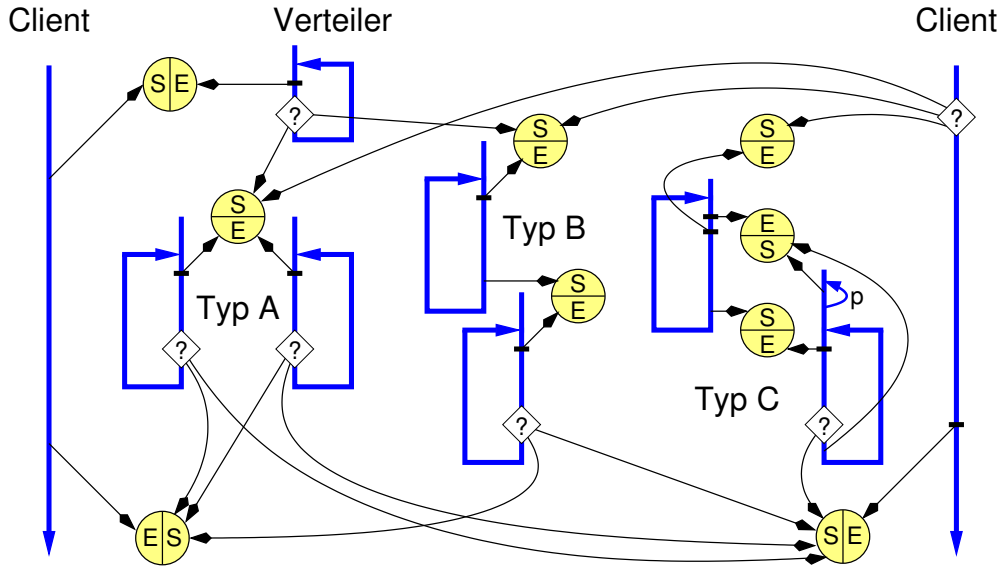
Aufgabe K7: Interaktionsmuster Team

Erstellen Sie Interaktionsdiagramme nach den folgenden Beschreibungen.

a) Ein Client beauftragt asynchron ein Team mit Verteiler. Das Team bearbeitet zwei Arten von Aufträgen. Aufträge vom Typ A übernimmt ein mit zwei Prozessen reproduzierter Server. Für Aufträge vom Typ B ist ein zweistufiges, offenes Fließband zuständig.



b) Das Team aus Teilaufgabe a wird um einen dritten Auftragstyp C ergänzt, den der Verteiler nicht kennt. Ein zweiter Client schickt mit einer neuen Bibliothek, welche ohne Verteiler arbeitet, einen asynchronen Auftrag an das Team. Aufträge vom Typ C bearbeitet ein Erzeuger–Verbraucher–System, das höchstens p Aufträge gleichzeitig annimmt. Es wird vom Verbraucher initialisiert.



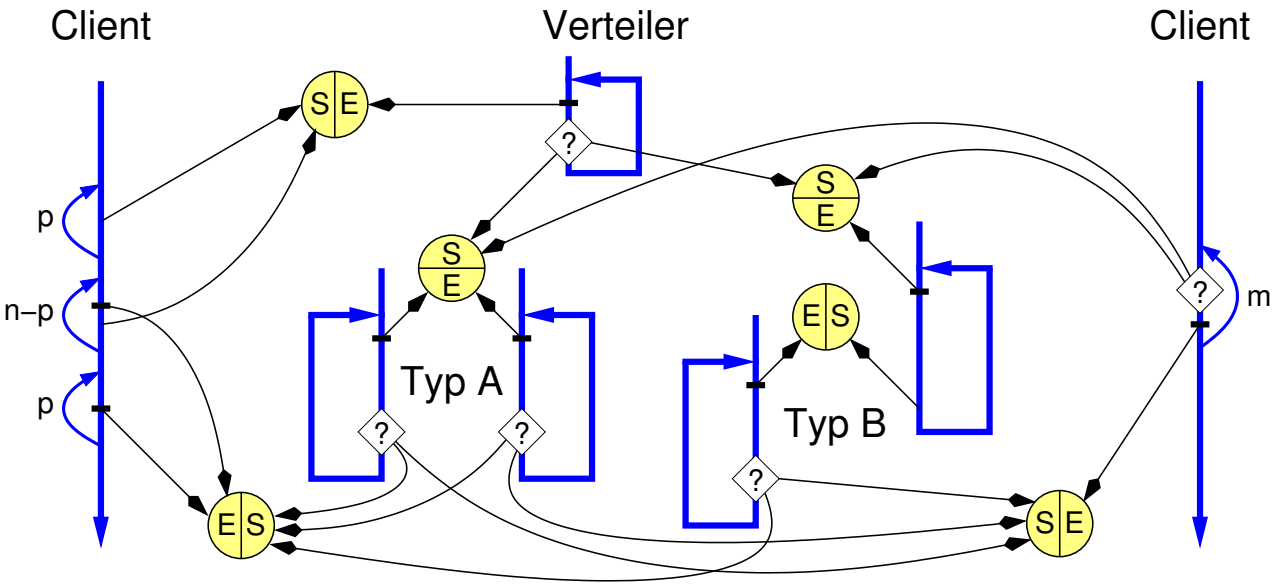
Ein Pfeil vom Verbraucher zum Antwortkanal des ersten Clients wäre auch nicht falsch.

Aufgabe K8: Interaktionsdiagramm

8 Pkt

Erstellen Sie ein Interaktionsdiagramm nach der folgenden Beschreibung:

Ein Client (*links*) sendet n Aufträge mit p-facher Pufferung an ein Team mit Verteiler und empfängt die Antworten. Das Team kennt zwei Auftragstypen. Typ A wird von zwei reproduzierten Prozessen bearbeitet, Typ B von einem offenen, zweistufigen Fließband. Ein anderer Client (*rechts*) sendet m Aufträge an das gleiche Team, aber ohne Verteiler. Dieser Client empfängt die Antwort direkt nach dem Senden eines Auftrags.



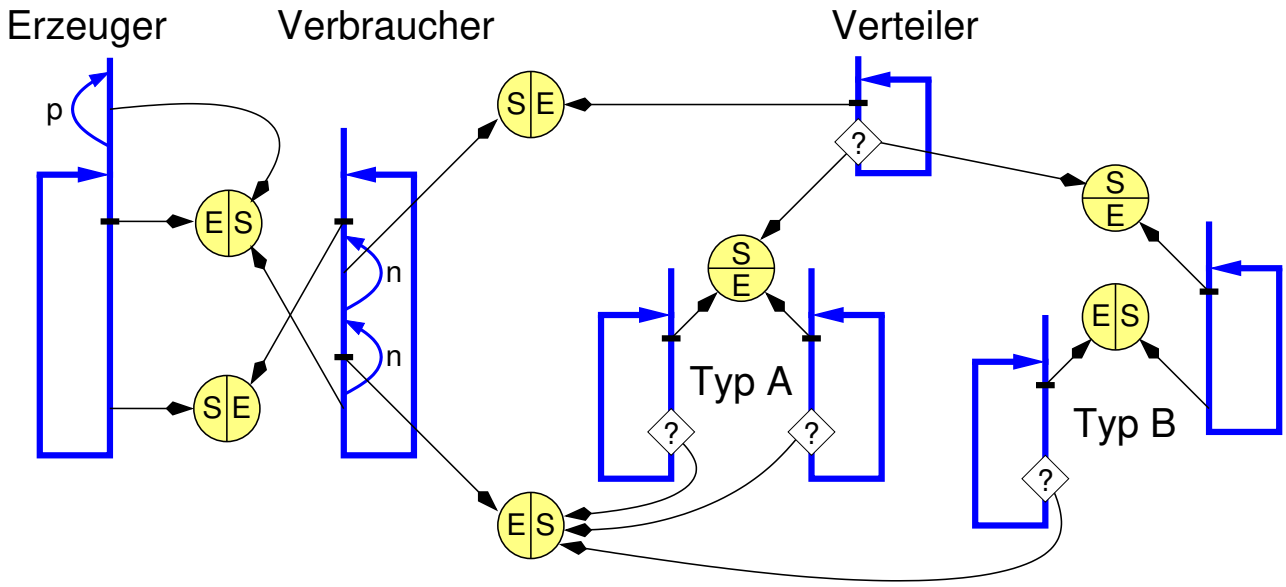
Punkte: Pufferung 2, Team 2, Verteiler 1, Repro 1, Fließband 1, Schleife ohne Verteiler 1

Aufgabe K9: Interaktionsdiagramm

8 Pkt

Erstellen Sie ein Interaktionsdiagramm nach der folgenden Beschreibung:

In einem Erzeuger/Verbraucher–System zirkulieren p Nachrichten. Die Initialisierung übernimmt der Erzeuger. Während der Bearbeitung einer Nachricht vergibt der Verbraucher n Aufträge mit unbeschränkter Pufferung und empfängt die Antworten. Diese Aufträge bearbeitet ein Team mit Verteiler. Es kennt zwei Auftragstypen. Typ A wird von zwei reproduzierten Prozessen bearbeitet, Typ B von einem zweistufigen Fließband.



Punkte: E/V 2, Init. 1, unbeschr. Pufferung 1, Team mit V. 2, Repro 1, Fließband 1

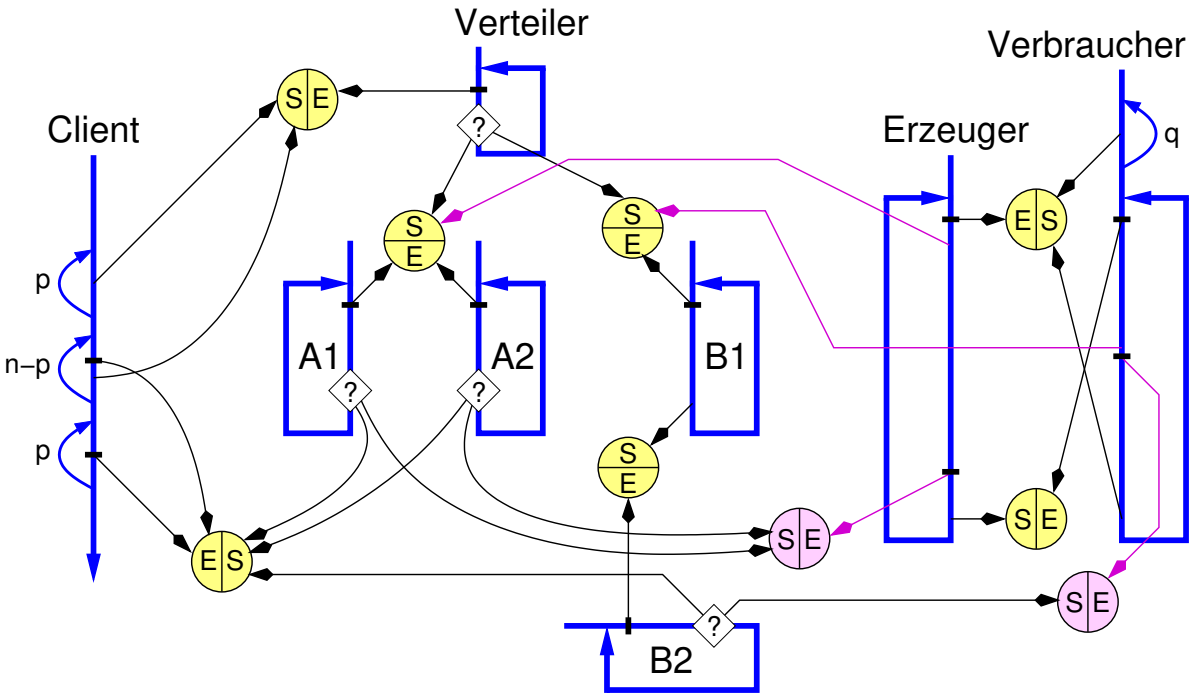
Aufgabe K10: Interaktionsdiagramm

8 Pkt

Erstellen Sie ein Interaktionsdiagramm nach der folgenden Beschreibung:

Ein Team mit Verteiler bearbeitet zwei Arten von Aufträgen, A und B. Die Aufträge vom Typ A übernehmen insgesamt zwei reproduzierte Prozesse. Für Aufträge vom Typ B ist ein zweistufiges, offenes Fließband zuständig. Ein Client schickt n Aufträge mit beschränkter Pufferung und Limit p an den Verteiler.
Ein Erzeuger/Verbraucher–System schickt ebenfalls Aufträge an das Team, aber ohne den Verteiler. Der Erzeuger schickt asynchron nur Aufträge vom Typ A, der Verbraucher synchron nur Aufträge vom Typ B. Beide senden und empfangen während ihrer Arbeitsphase je einmal. Im Erzeuger/Verbraucher–System zirkulieren q Nachrichten. Die Initialisierung übernimmt der Verbraucher.

Zeichenvorschlag: Team in der Mitte, Client links, Erzeuger/Verbraucher rechts.



Punkte:

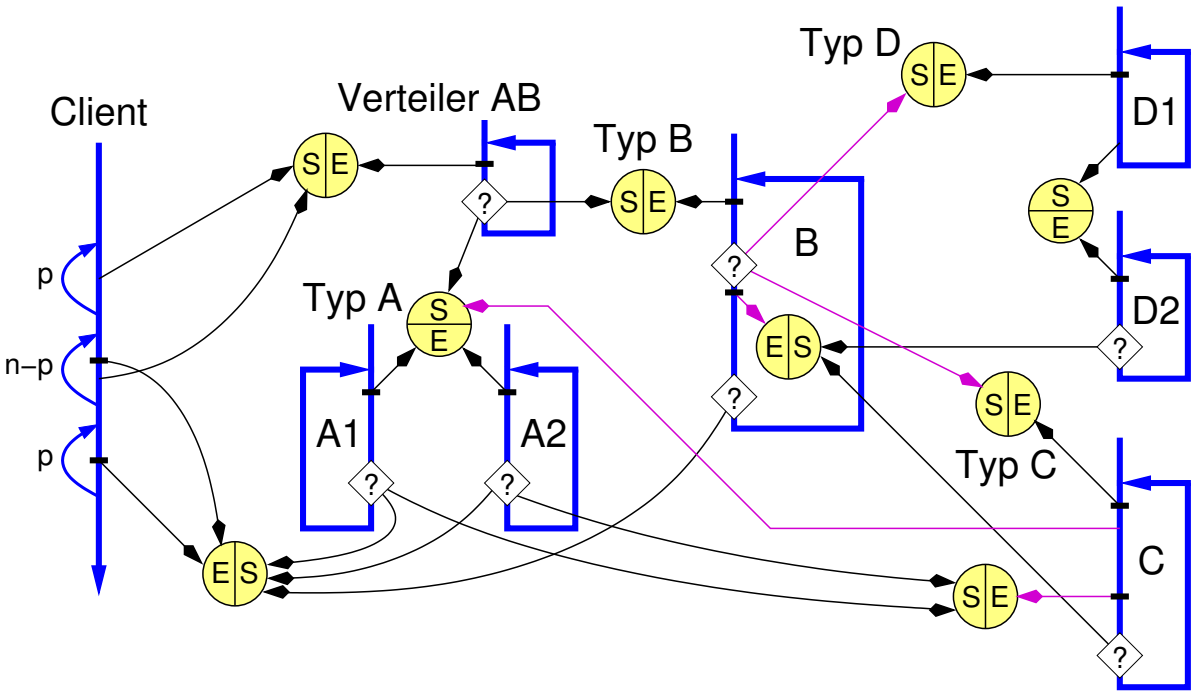
1.5	Pufferung beschränkt
1.5	Team mit Verteiler
1	Reproduktion
1	Fließband
2	Erzeuger/Verbraucher mit Initialisierung
1	Aufträge am Verteiler vorbei

Aufgabe K11: Interaktionsdiagramm

8 Pkt

Erstellen Sie ein Interaktionsdiagramm nach der folgenden Beschreibung:

Ein Client schickt n Aufträge mit beschränkter Pufferung und Limit p an ein Team mit Verteiler. Dieses Team kennt zwei Auftragstypen, A und B. Aufträge vom Typ A bearbeiten insgesamt zwei reproduzierte Prozesse.
Aufträge vom Typ B übernimmt ein einzelner Prozess. Während seiner Arbeitsphase schickt er jeweils einen synchronen Auftrag an ein weiteres Team. Das zweite Team arbeitet ohne Verteiler. Es kennt zwei Auftragstypen, C und D. Aufträge vom Typ D übernimmt ein offenes Fließband mit zwei Stufen.
Aufträge vom Typ C bearbeitet ein einzelner Prozess. Während seiner Arbeitsphase schickt er jeweils einen asynchronen Auftrag und empfängt die Antwort. Dieser Auftrag ist vom Typ A und geht an das erste Team, ohne dessen Verteiler zu benutzen.



Punkte:

2	Pufferung beschränkt
1.5	Team mit Verteiler
1	Reproduktion
1.5	Team ohne Verteiler, synchroner Auftrag
1	Fließband
1	asynchroner Auftrag am Verteiler vorbei

Aufgabe K12: Zelle

Ein Kernobjekt vom Typ *Zelle* ist entweder leer oder enthält einen Wert. Die Operation `give` übergibt einen Wert an die Zelle. Enthält sie schon einen Wert, wird dieser überschrieben. Die Operation `take` nimmt den Wert aus einer Zelle und leert sie damit. Ist die Zelle bereits leer, muss der Aufrufer warten, bis er einen neu übergebenen Wert erhält. Die Implementierung des Kernobjekts verwendet folgende Attribute:

Flag, ob die Zelle einen Wert enthält. Initial gelöscht, d.h. Zelle leer.

Inhalt, der Wert des letzten Aufrufs von `give`. Nur gültig wenn das Flag gesetzt ist. Der initiale Wert spielt deshalb keine Rolle.

Wartemenge für Prozesse. Initial leer.

a) Welche ungültigen Fälle müssen ausgeschlossen werden?

1. **Flag** ist gesetzt, aber die **Wartemenge** nicht leer.
   Das heißt, ein Prozess wartet, obwohl ein Wert in der Zelle steht. +1
2. **Flag** ist gesetzt, aber in **Inhalt** steht etwas anderes als der zuletzt an `give` übergebene Wert. +1

b) Beschreiben Sie den Ablauf von `give`. Aufrufargument ist der Wert. 4 Pkt

- **Wartemenge** leer? +1

ja:
- übergebenen Wert als **Inhalt** merken +0.5
- **Flag** setzen, d.h. die Zelle ist gefüllt +0.5

nein:
- einen Prozess aus der **Wartemenge** nehmen +0.5
- übergebenen Wert als Rückgabewert des Prozesses setzen +1
- den Prozess deblockieren +0.5

c) Beschreiben Sie den Ablauf von `take`. Ergebnis des Aufrufs ist der Wert. 4 Pkt

- **Flag** gesetzt? +1

ja:
- **Inhalt** als Rückgabewert des Aufrufs setzen +1
- **Flag** löschen, d.h. die Zelle ist leer +1

nein:
- Aufrufer blockieren und in **Wartemenge** stecken +1

Aufgabe K13: Rendezvous mit Wertaustausch

Ein Kernobjekt vom Typ *Rendezvous* ermöglicht je zwei Prozessen, sich zu synchronisieren und dabei einen Wert auszutauschen. Es bietet eine Operation `rendezvous` an, die synchron arbeitet. Sie erwartet einen Wert als Argument und gibt einen Wert als Ergebnis zurück. Der erste Prozess, der die Operation aufruft, muss auf den zweiten warten. Dann erfolgt der Austausch der Werte. Danach ist das Kernobjekt wieder im Ausgangszustand, bereit für das nächste Rendezvous.

a) Welche Attribute benötigt das Kernobjekt? Wie sind sie initialisiert? Welche ungültigen Fälle müssen ausgeschlossen werden?

- Zeiger **WP** auf einen wartenden Prozess. Initial `null`.
  alternativ: Wartemenge, initial leer, in der höchstens ein Prozess liegen wird
- Wert **WW**, der vom wartenden Prozess übergeben wurde. Nur gültig, wenn **WP** nicht `null`. Der initiale Wert spielt deshalb keine Rolle.

Ein ungültiger Fall wäre, dass mehr als ein Prozess wartet. Mit einem Zeiger statt einer Wartemenge ist das nicht darstellbar. Trotzdem muss man darauf achten, keinen Prozess zu blockieren, wenn bereits einer wartet.

Ein weiterer ungültiger Fall wäre, dass ein Prozess wartet, aber in **WW** nicht der von ihm übergebene Wert steht.

Wenn man den Wert aus der Rechenumgebung des wartenden Prozesses holt, ist **WW** *überflüssig. Dieses Thema bespreche ich aber erst später im Semester.*

b) Beschreiben Sie den Ablauf von `rendezvous`. Argument ist ein Aufrufwert. Zudem gibt es einen Rückgabewert. 4 Pkt

- **WP** `null`? *bzw. Wartemenge leer?* +1

ja: *Aufrufer muss warten*

- Aufrufer blockieren und in **WP** stecken *bzw. in Wartemenge* +1
- Aufrufwert in **WW** merken +1

nein: *Rendezvous des Aufrufers mit dem wartenden Prozess*

- **WW** als Rückgabewert des Aufrufs setzen
- Aufrufwert als Rückgabewert des Prozesses in **WP** setzen +1
- den Prozess in **WP** deblockieren +0.5
- **WP** auf `null` setzen +0.5
  bzw. schon oben: Prozess aus Wartemenge nehmen

Das beschriebene Verhalten entspricht in etwa dem von `rendezvous` in Plan 9:
`http://man.cat-v.org/plan_9/2/rendezvous`

Aufgabe K14: Semaphor mit Zusatzmethoden

Ausgangspunkt ist ein als Kernobjekt implementiertes Semaphor mit den Operationen `P` und `V`, wie im Begleitbuch beschrieben. Die Methoden heißen hier `Psyn` und `Vasyn`.

Attribute:

- **Zähler**, nicht negativ. Initialwert beim Erzeugen anzugeben.
- **Wartemenge** für Prozesse. Initial leer.

Dateninvariante:

- Wenn **Zähler** > 0, dann **Wartemenge** leer.
  (d.h. ungültiger Fall: **Zähler** > 0 und **Wartemenge** nicht leer)

Methoden:

`Psyn`: Runterzählen um 1, synchron, wartet bei Zählerstand 0

- **Zähler** > 0 ?

ja: **Zähler** vermindern um 1

nein: Aufrufer blockieren und in **Wartemenge** stecken

`Vasyn`: Hochzählen um 1, asynchron

- **Wartemenge** leer?

ja: **Zähler** erhöhen um 1

nein: Prozess aus **Wartemenge** nehmen und deblockieren

Die Implementierung soll um die unten aufgeführten Methoden ergänzt werden. Welche davon lassen sich einfach, d.h. ohne Änderungen an den Attributen oder der Dateninvariante, implementieren? Beschreiben Sie für diese jeweils den Ablauf. Begründen Sie bei den anderen, warum keine einfache Implementierung möglich ist.

a) `Ptry`: Versuchendes Runterzählen um 1. Erfolg falls Zähler > 0.

- **Zähler** > 0 ? +1

ja: **Zähler** vermindern um 1; Erfolg melden +1.5

nein: Misserfolg melden +0.5

b) `Vtry`: Versuchendes Hochzählen um 1. Erfolg falls ein Prozess wartet.

Die Beschreibung ist etwas widersinnig, weil diese Methode niemals hochzählt. Sie hat nur Erfolg, wenn ein Prozess wartet, der das Hochzählen ausgleicht.

- **Wartemenge** leer? +1

ja: Misserfolg melden +0.5

nein: Prozess aus **Wartemenge** nehmen und deblockieren; Erfolg melden +1.5

c) `PNsyn`: Runterzählen um mehr als 1, synchron. Argument N, positive Zahl.

Das geht nicht ohne tiefgreifende Änderungen. Die Dateninvariante gilt nicht mehr, weil trotz Zähler > 0 ein Prozess in der Wartemenge liegen kann, der um mehr als den Zählerstand runterzählt. In der Wartemenge können sich mehrere Prozesse befinden, die unterschiedlich weit runterzählen. Das erfordert Änderungen an allen Methoden, die Prozesse aus der Wartemenge deblockieren.

Aufgabe K15 beschäftigt sich mit dieser Problemstellung.

d) `VNasyn`: Hochzählen um mehr als 1, asynchron. Argument N, positive Zahl.

- Delta := N
- solange Delta > 0 und **Wartemenge** nicht leer: +1.5
  - Prozess aus **Wartemenge** nehmen und deblockieren +1
  - Delta vermindern um 1 +0.5
- **Zähler** um Delta erhöhen +1

e) `PNtry`: Versuchendes Runterzählen um mehr als 1. Erfolg falls Zähler groß genug.

Im Gegensatz zu `PNsyn` ist diese Methode einfach zu implementieren, weil der Aufruf nicht gespeichert und später zu Ende gebracht werden muss.

- **Zähler** $\geq N$? +1

ja: **Zähler** vermindern um N; Erfolg melden +1.5

nein: Misserfolg melden +0.5

f) `VNtry`: Versuchendes Hochzählen um mehr als 1. Erfolg falls mindestens so viele Prozesse warten, dass sie um N runterzählen.

Da alle Prozesse in der Wartemenge um 1 runterzählen, muss die Methode die Anzahl der Elemente in der Wartemenge abfragen können. Falls das möglich ist:

- **Wartemenge** enthält weniger als N Prozesse? +1

ja: Misserfolg melden +0.5

nein: N Prozesse aus **Wartemenge** nehmen und deblockieren; Erfolg melden +1.5

Die Semantik dieser Methode ergibt meines Erachtens keinen Sinn. Ich habe sie nur wegen der Symmetrie mit in die Aufgabe genommen.

g) `P0asyn`: Runterzählen bis 0.

- **Zähler** auf 0 setzen +1

h) `V0asyn`: Hochzählen bis Wartemenge leer.

Die Beschreibung ist etwas widersinnig, weil diese Methode niemals hochzählt.

- alle Prozesse aus **Wartemenge** nehmen und deblockieren +1

Aufgabe K15: Semaphor mit Argumenten

Ein erweitertes Semaphor ist als Kernobjekt implementiert. Die Operationen zum Hoch- und Runterzählen erhalten jeweils ein Argument, um wieviel der Zählerstand verändert werden soll. Das Kernobjekt bietet folgende Operationen an:

`decrementSyn`: Runterzählen synchron. Argument ist eine positive Zahl R.

`decrementTry`: Runterzählen versuchend. Argument ist eine positive Zahl R.
Rückgabewert ist entweder Erfolg oder Misserfolg.

`incrementAsyn`: Hochzählen asynchron. Argument ist eine positive Zahl H.

Die Implementierung des Kernobjekts verwendet folgende Attribute:

Z: Zählerstand, nicht negativ. Initial wählbar.

WR: Wartemenge für Runterzähler. Initial leer.

Aufrufe der `decrement`–Operationen werden reihenfolgetreu durchgeführt. Ein Aufrufer darf also keinen anderen, noch blockierten Aufrufer überholen.

a) Welche Strategie verwendet die Wartemenge WR? Welche zusätzlichen Funktionen muss WR unterstützen, gegenüber einem Semaphor ohne Argumente R und H?

WR verwendet eine Reihenfolgestrategie (FIFO, engl.: *first in — first out*), sonst könnten sich Aufrufer überholen. Für jeden wartenden Prozess muss auch die Zahl R gespeichert sein, um die er runterzählt. Die Elemente der Menge sind also nicht Prozesse P, sondern Tupel (P,R). Die Zahl R für den ersten Prozess muss abfragbar sein, ohne das Tupel aus der Menge zu nehmen.

b) Welches sind die ungültigen Fälle des Kernobjekts?

Es gibt nur einen ungültiger Fall:

- WR nicht leer und das erste R darin ist kleiner oder gleich Z.

Das heißt, der erste wartende Prozess könnte Z weit genug runterzählen.

c) Beschreiben Sie den Ablauf von `decrementSyn` mit Argument R.

Den Zählerstand zu prüfen reicht nicht. Der Aufrufer darf auch nicht überholen. Es ist nicht verlangt, den Wertebereich von R zu kontrollieren, aber guter Stil.

- falls R ≤ 0 : Abbruch mit Fehler
- WR leer und Z $\geq$ R ? +2

ja: – Z um R runterzählen +1

nein: – Aufrufer blockieren +0.3

– (Aufrufer,R) in WR legen +0.7

d) Beschreiben Sie den Ablauf von `incrementAsyn` mit Argument H.

H muss zum aktuellen Zählerstand addiert werden, um zu entscheiden, ob einer oder mehrere der wartenden Prozesse deblockieren.

- falls H ≤ 0 : Abbruch mit Fehler *guter Stil, siehe oben*
- Z um H hochzählen +1
- solange WR nicht leer und Z $\geq$ R aus WR: +2
  - (Prozess,R) aus WR nehmen +0.7
  - Z um R runterzählen +1
  - Prozess deblockieren +0.3

e) Beschreiben Sie den Ablauf von `decrementTry` mit Argument R.

Die Fallunterscheidung ist die gleiche wie bei `decrementSyn`. Hier wird aber in beiden Fällen ein Ergebnis zurückgegeben, entweder Erfolg oder Misserfolg.

- falls R ≤ 0 : Abbruch mit Fehler *guter Stil, siehe oben*
- WR leer und Z $\geq$ R ? +2

ja:
- Z um R runterzählen +1
- Erfolg zurückgeben +0.5

nein:
- Misserfolg zurückgeben +0.5

Achtung: Misserfolg ist *kein* Abbruch oder Fehler, sondern einer von zwei möglichen, korrekten Abläufen. Implementierungen verwenden vielleicht den gleichen Mechanismus, um Fehler und Misserfolg an den Aufrufer zurückzugeben. Zum Beispiel erlaubt C nur einen einzigen direkten Rückgabewert. Meist bedeutet dann 0 den fehlerfreien Erfolg. Bei anderen Werten muss aber der Aufrufer unterscheiden, ob es sich um den eingeplanten Misserfolg oder um einen tragischen Fehler handelt.

f) Statt der Reihenfolge soll eine andere Strategie zum Einsatz kommen, die Aufrufer mit niedrigem R bevorzugt. Wo und wie ändern sich dabei die Abläufe?

Bei `incrementAsyn` ändert sich nichts. Die `decrement`–Operationen ändert sich für den Fall, dass die Wartemenge nicht leer ist. Falls der Aufrufer beim Einfügen in die Wartemenge an deren Spitze gelangen würde, darf er trotz der Wartenden runterzählen, sofern der Zählerstand hoch genug ist. Diese Möglichkeit ist auch bei `decrementTry` zu prüfen, obwohl der Aufrufer dort niemals blockiert.

g) Statt der bisherigen soll eine Strategie zum Einsatz kommen, welche die Prioritäten der aufrufenden Prozesse berücksichtigt. Was ist zu bedenken?

Prioritäten können sich ändern, während Prozesse in der Wartemenge stecken. Wenn sich die Reihenfolge in der Wartemenge ändert, muss neu geprüft werden, ob nun Runterzähler deblockiert werden dürfen.

Aufgabe K16: Client/Server–Kanal

Ein spezieller Kanal für synchrone Aufträge an Server bietet folgende Operationen an:

`callSyn`: Ein Client sendet einen Auftrag und wartet auf ein Ergebnis.

`takeSyn`: Ein Server nimmt einen Auftrag entgegen.

`replyAsyn`: Ein Server beantwortet einen Auftrag.

Der Kanal verwendet ausschließlich die Referenzablage. Beim Zustellen eines Auftrags an einen Server generiert er eine Auftragskennung. Diese Kennung ist Rückgabewert von `takeSyn` und Argument von `replyAsyn`. Die Implementierung verwendet folgende Datenstrukturen:

CmA: Wartemenge für Clients mit Auftrag. Initial leer.
Enthält Tupel aus (Client, Nachrichtenreferenz, Antwortpufferreferenz).

SoA: Wartemenge für Server ohne Auftrag. Initial leer.
Enthält Tupel aus (Server, Auftragspufferreferenz).

CoE: Wartemenge bzw. Tabelle für Clients ohne Ergebnis. Initial leer.
Enthält Tupel aus (Kennung, Client, Antwortpufferreferenz).

In CoE speichert der Kanal, welcher Client auf das Ergebnis für welche Kennung wartet. Der Zugriff auf die Elemente erfolgt über die Kennung.

Diese Art von Kanal gibt es zum Beispiel in QNX Neutrino:
`https://www.qnx.com/developers/docs/7.0.0/#com.qnx.doc.neutrino.sys_arch/topic/ipc_Sync_messaging.html`

a) Beschreiben Sie den Ablauf der Client–Operation `callSyn`. Argumente sind die Referenzen auf eine Auftragsnachricht und einen Empfangspuffer für die Antwort.

Falls ein Server wartet, wird ihm der Auftrag mit einer generierten Kennung zugestellt und der Client in CoE gesteckt. Ansonsten landet der Client in CmA. In beiden Fällen blockiert also der aufrufende Client.

- Aufrufer, d.h. Client blockieren +1
- SoA enthält Element? +1

Ja:
- (Server, Auftragspufferreferenz) aus SoA nehmen +0.7
- Kennung generieren, als Rückgabewert für Server setzen +1
- (Kennung, Client, Antwortpufferreferenz) in CoE stecken +1
- Auftragsnachricht in Auftragspuffer des Servers kopieren +1
- Server deblockieren +0.3

Nein:
- (Client, Nachrichtenreferenz, Anwortpufferreferenz) in CmA stecken +1

b) Beschreiben Sie den Ablauf der Server–Operation `takeSyn`. Argument ist die Referenz auf einen Empfangspuffer für den Auftrag, Rückgabewert ist die Auftragskennung.

Falls ein Client in CmA wartet, wird dessen Auftrag abgeholt und eine Kennung dafür generiert. Der Client wandert von CmA nach CoE. Ansonsten landet der Server in SoA.

- CmA enthält Element? +1

Ja:
- (Client, Nachrichtenreferenz, Anwortpufferreferenz) aus CmA nehmen +0.5
- Kennung generieren, als Rückgabewert für Aufrufer setzen +1
- (Kennung, Client, Anwortpufferreferenz) in CoE stecken +0.5
- Auftragsnachricht in Empfangspuffer des Aufrufers kopieren +1

Nein:
- Aufrufer, d.h. Server blockieren +0.3
- (Server, Auftragspufferreferenz) in SoA stecken +0.7

Man könnte die Kennung auch immer in `callSyn` generieren. Dann muss sie aber mit in CmA gespeichert werden, was laut Aufgabenstellung nicht der Fall ist.

c) Beschreiben Sie den Ablauf der Server–Operation `replyAsyn`. Argumente sind die Auftragskennung und eine Referenz auf die Antwortnachricht.

Bei einem gültigen Aufruf wartet der passende Client bereits in CoE, das Ergebnis kann also direkt zugestellt werden. Falls der Server eine ungültige Kennung übergibt oder den Auftrag bereits vorher beantwortet hat, fehlt der passende Client.

- (*Auftragskennung*, Client, Anwortpufferreferenz) aus CoE nehmen +0.7
  - Kennung nicht gefunden? ⇒ Abbruch mit Fehler +1
- Ergebnis in Antwortpuffer des Clients kopieren +1
- Client deblockieren +0.3

Aufgabe K17: Kanal mit Durchschlag

Ein spezieller Kernobjekttyp „Durchschlagskanal“ stellt Nachrichten zweimal zu: erst als Original an einen Empfänger, dann als Durchschlag an einen Auditor. Der Auditor erhält zusätzlich die Kennungen von Sender und Empfänger. Um Manipulationen des Durchschlags zu verhindern, wird jede Nachricht zuerst in einen beschränkten Kanalpuffer kopiert. Ist der Puffer voll, scheitert das Senden. Dieser Kanaltyp bietet folgende Operationen an:

`sendAsyn`: sendet eine Nachricht, ohne zu warten

`receiveSyn`: empfängt eine Nachricht im Original, wartet bis zum Eintreffen

`receiveAuditSyn`: empfängt eine Nachricht als Durchschlag, wartet bis zum Eintreffen

Die Implementierung verwendet folgende Datenstrukturen, initial sind alle leer:

P: Pufferspeicher für Nachrichteninhalte

MO: Menge für Originale, also neue Nachrichten

MD: Menge für Durchschläge, also zugestellte Nachrichten

WE: Wartemenge für Empfänger von Originalen

WA: Wartemenge für Auditoren, also Empfänger von Durchschlägen

a) Welche Informationen speichern die Elemente der vier Mengen?
Hinweis: Die Antwort ergibt sich aus den Ablaufbeschreibungen.

MO: (Nachrichtenreferenz, Senderkennung)

MD: (Nachrichtenreferenz, Senderkennung, Empfängerkennung)

WE: (Empfangspufferreferenz, Empfängerprozess)

WA: (Auditpufferreferenz, Auditorprozess)

Nachrichten durchlaufen zwei Stationen, repräsentiert durch MO und MD. Auditoren erwarten sowohl Sender- als auch Empfängerkennung. In MO ist nur der Sender bekannt, in MD auch der Empfänger.

WE und WA sind die Wartemengen für blockierte Empfänger bzw. Auditoren. Sie verwenden laut den Teilaufgaben unterschiedliche Arten von Puffern. Ein Empfangspuffer kann nur die Nachricht aufnehmen, ein Auditpuffer zusätzlich die Kennungen von Sender und Empfänger. Man könnte für Auditoren auch gewöhnliche Empfangspuffer nehmen und zusätzliche Argumente für die Kennungen vorsehen.

Die Aufgabenstellung beschreibt nicht, welche Form die Kennungen von Sender und Empfänger haben. Eine Prozesskennung wäre in der Praxis für ein Audit–Log unbrauchbar, da sie sich im Nachhinein keiner Instanz und keinem Programm mehr zuordnen lässt. Das ist aber auch nicht Thema der Aufgabe.

b) Beschreiben Sie den Ablauf des Durchschlagempfangens. (`receiveAuditSyn`)
Argument ist die Referenz auf einen Auditpuffer.

Die einfachste der drei Operationen. Der Auditor erwartet ein Element in MD.

- MD leer? +1

ja:
- Aufrufer blockieren +0.3
- (Auditpufferreferenz, Aufrufer) in WA legen +0.7

nein: *abholen...*

- (Nachrichtenref., Sender-, Empfängerkennung) aus MD nehmen +1
- Nachricht aus P in Auditpuffer kopieren +1
- Sender- und Empfängerkennung in Auditpuffer kopieren +1
- Nachricht aus P löschen — *nicht vergessen!* +1

c) Beschreiben Sie den Ablauf des synchronen Empfangens. (`receiveSyn`)
Argument ist die Referenz auf einen Empfangspuffer.

Diese Operation kann zwei Ereignisse auslösen, nämlich das Zustellen an den aufrufenden Empfänger und das Zustellen des Durchschlags an einen Auditor. Der Empfänger erwartet ein Element in MO.

- MO leer? +1

ja:
- Aufrufer blockieren +0.3
- (Empfangspufferreferenz, Aufrufer) in WE legen +0.7

nein: *abholen...*

- (Nachrichtenreferenz, Senderkennung) aus MO nehmen +0.5
- Nachricht in Empfangspuffer kopieren +1
- Kennung des Empfängers, also des Aufrufers, bestimmen +0.5
- WA leer? +1

ja: *kein Auditor, Durchschlag muss warten*
- (Nachrichtenref., Sender-, Empfängerkennung) in MD legen +1
- Ende des Aufrufs

nein: *zustellen...*

- (Auditpuffer, Auditor) aus WA nehmen +0.5
- Nachricht in Auditpuffer kopieren +1
- Sender- und Empfängerkennung in Auditpuffer kopieren +1
- Auditor deblockieren +0.5
- Nachricht aus P löschen — *nicht vergessen!* +1

d) Beschreiben Sie den Ablauf des asynchronen Sendens. (`sendAsyn`)
Argument ist die Referenz auf eine Nachricht im Adressraum des Senders.

Auch diese Operation kann zwei Ereignisse auslösen. Das Zustellen eines Durchschlags an den Auditor verläuft exakt wie beim entsprechenden Fall in `receiveSyn`.

Laut Aufgabenstellung wird die Nachricht auf jeden Fall in einen Kanalpuffer kopiert, auch wenn ein Empfänger bereitsteht. Das verhindert, dass weitere Prozesse im Adressraum von Sender oder Empfänger die Nachricht verändern, nachdem sie vom Kanal abgeholt wurde, aber bevor der Auditor einen Durchschlag erhält. Natürlich muss man dazu auch immer die Kopie aus dem Kanalpuffer übergeben.

- Platz für Nachricht in P? +0.5

nein: Abbruch mit Fehler +0.5

- Nachricht in P kopieren, neue Referenz NPRef auf Nachricht in P +1
- Kennung des Senders, also des Aufrufers, bestimmen +0.5

- WE leer? +1

ja: *kein Empfänger, Nachricht muss warten*

  – (NPRef, Senderkennung) in MO legen +1

  – Ende des Aufrufs

nein: *zustellen...*

- (Empfangspuffer, Empfänger) aus WE nehmen +0.7
- Kennung des Empfängers bestimmen +0.5
- Nachricht aus P in Empfangspuffer kopieren +1
  wichtig: aus P, nicht aus dem Adressraum des Senders
- Empfänger deblockieren +0.3

- WA leer? +1

ja: *kein Auditor, Durchschlag muss warten*

  – (NPRef, Senderkennung, Empfängerkennung) in MD legen +1

  – Ende des Aufrufs

nein: *zustellen...*

- (Auditpuffer, Auditor) aus WA nehmen +0.5
- Nachricht aus P in Auditpuffer kopieren +1
  wichtig: aus P, nicht aus dem Adressraum des Senders oder Empfängers
- Sender- und Empfängerkennung in Auditpuffer kopieren +1
- Auditor deblockieren +0.5
- Nachricht aus P löschen — *nicht vergessen!* +1

Aufgabe K18: Prozessverwaltung (Nachlese)

Die Zuteilung von Rechenzeit durch den Aufgreifer (engl.: *scheduler*) ist eng verzahnt mit der Verwaltung von Prozessen in der Bereitmenge. Der Aufgreifer muss schnell bestimmen, welcher Prozess als nächstes Rechenzeit erhalten soll. Dazu ordnet er die bereitstehenden Prozesse nach Prioritäten. Eine Tabelle in der englischsprachigen Wikipedia nennt die Algorithmen, die in verschiedenen Betriebssystemen zum Einsatz kommen:

`https://en.wikipedia.org/wiki/Scheduling_(computing)#Summary`

Besonders häufig ist die *multilevel feedback queue* (MLFQ) aufgeführt. Lesen Sie nach, was es damit auf sich hat und beantworten Sie die folgenden Fragen.

Für den Detailgrad dieser Aufgabe genügt die in der Tabelle verlinkte Seite:
`https://en.wikipedia.org/wiki/Multilevel_feedback_queue`

a) Welche Auszeichnung erhielt der Erfinder der MLFQ?

Fernando J. Corbató erhielt 1990 den Turing–Award.
`https://amturing.acm.org/award_winners/corbato_1009471.cfm`

b) In welcher Größenordnung liegt die Anzahl der *Level*?

Die in der Aufgabenstellung verlinkte Wikipedia–Seite nennt vor der Tabelle einige Beispiele. Konkrete Zahlen liegen im Wertebereich eines Byte, zum Beispiel 32 in Windows oder 256 in OpenBSD. Es geht also um dutzende bis hunderte Level.

c) Woher kommt das *Feedback*? Wie wirkt es sich aus?

Das Feedback beruht auf dem Verhalten jedes einzelnen Prozessen. Prozesse, die ihre Zeitscheiben aufbrauchen, wandern in Level mit niedrigerer Priorität. Prozesse, die vor Ablauf ihrer Zeitscheibe blockieren, wandern in Level mit höherer Priorität. Das soll Prozesse bevorzugen, die nur kurz auf Ereignisse (I/O) reagieren und schnell wieder blockieren. Bei diesen ist die Reaktionszeit wichtig. Bei Prozessen die lange rechnen spielen Reaktionszeiten eine geringere Rolle.

d) Vergleichen Sie die MLFQ mit der Datenstruktur aus Aufgabe V2 „Bereitmenge und Aufgreifstrategie“.

Aufgabe V2 verwendet eine *multilevel queue* mit drei Leveln, ohne Feedback.

e) Eignet sich die MLFQ auch für andere Situationen, in denen Prozesse verwaltet werden? Zum Beispiel bei blockierten Prozessen in einer Wartemenge?

Nein. Das Sortieren von Prozessen nach Prioritäten ist auch in anderen Situationen sinnvoll. Aber das *Feedback* gibt es nur beim Aufgreifer. Die Datenstruktur ist anderswo keine ML*F*Q mehr. Ob separate Queues für dutzende bis hunderte Prioritätsstufen ohne Feedback noch sinnvoll sind, hängt von der Anzahl der Prozesse und der Häufigkeit des Einfügens ab. Eher nicht.

Aufgabe K19: Prozessmodell (Recherche)

Recherchieren Sie das *five-state process model* und vergleichen Sie es mit dem Prozessmodell aus der Vorlesung.

Im Modell von Stallings heißen die fünf Zustände New, Ready, Running, Blocked und Exit. Die übliche grafische Darstellung zeigt einen Sägezahn nach unten. Eine schönere Variante mit symmetrischer Darstellung benennt zwei Zustände anders: Waiting statt Blocked und Terminated statt Exit.
Nicht gemeint ist das *five-state process management model* mit zusätzlichen Zuständen Suspend Ready und Suspend Blocked (oder ähnlich, je nach Quelle). Bei diesem fehlen New und Exit. Beide sind aber durch offene Pfeile angedeutet. Die vollständige Version dieses Modells ist das *seven-state process model* und somit hier nicht gefragt.

a) Welche Zustände entsprechen direkt einem im Prozessmodell der Vorlesung?

Ready ≡ Bereit, Running ≡ Rechnend, Blocked ≡ Blockiert, Exit ≡ Fertig

b) Wie passen die restlichen Zustände zum Prozessmodell der Vorlesung?

New ist ein Fall von Inaktiv. Der Zustand Inaktiv (ob mit oder ohne Fertig) schließt aber noch weitere Fälle ein, die im *five-state process model* fehlen.

c) Welches Verhalten von Prozessen bildet das Prozessmodell aus der Vorlesung ab, das *five-state process model* (5SPM) aber nicht?

- Zwischenzeitliches Anhalten durch Deaktivieren und Aktivieren. Das 5SPM aktiviert nur neue Prozesse und deaktiviert nur am Ende eines Prozesses.
- Abschuss durch Deaktivieren aus einem beliebigen Zustand. Würde man diese Übergänge im 5SPM ergänzen, dann entspräche Exit nicht mehr Fertig, oder man bräuchte ein 6SPM.
- Erzeugen und Löschen sind im 5SPM nicht enthalten.

Aufgabe K20: Links in Dateisystemen (Recherche)

Das POSIX–Programm `ln` kann zwei verschiedene Arten von Links in Unix–Dateisystemen erzeugen. Das Windows–Programm `mklink.exe` kann drei verschiedene Arten von Links (und sogenannte Junctions) in NTFS–Dateisystemen erzeugen. Recherchieren Sie die verschiedenen Arten von Links.

A) `ln -s` *symbolischer Link*

B) `ln` *Hard Link*

C) `mklink` *symbolischer Link auf Datei*

D) `mklink /d` *symbolischer Link auf Verzeichnis*

E) `mklink /h` *Hard Link*

Den Begriff Soft Link verwendet man im POSIX–Umfeld gleichbedeutend mit symbolischen Links. Bei NTFS bezeichnet er aber Junctions, die sich sowohl von Hard Links als auch von symbolischen Links unterscheiden.

Welche der folgenden Aussagen gelten für welche Arten von Links?

1. Kann auf eine Datei zeigen. *A B C E*
2. Kann auf ein Verzeichnis zeigen. *A D*
3. Kann brechen, d.h. ins Leere zeigen. *A C D*
4. Kann in ein anderes Dateisystem (Speichermedium) zeigen. *A C D*
5. Der Link funktioniert weiter, wenn man das Ziel verschiebt. *B E*
6. Das Ziel kann sich ändern, wenn man den Link verschiebt. *A C D*

	A	B	C	D	E
1	+	+	+		+
2	+			+	
3	+		+	+	
4	+		+	+	
5		+			+
6	+		+	+	

Aufgabe K21: Dateien und Verzeichnisse

Die Aufgabenstellung zeigt drei Dateisysteme mit insgesamt acht Dateien. Das zweite ist beim ersten mit dem Pfad `/permissive` eingehängt, das dritte beim zweiten unter `/permissive/funny`. Die Dateiinhalte sind Namen von Software–Lizenzen, zum Beispiel GPL oder BSD. Geben Sie Pfade ohne Umwege an, also ohne überflüssige `../` oder `./`

a) Welche absoluten Pfade führen zu den acht Dateien?

CC-0:	*nicht erreichbar, verdeckt durch das eingehängte Dateisystem*
GPL:	`/political/left`
Unlicense:	`/political/anti` *nur so, der zweite Pfad ist verdeckt*
BSD:	`/permissive/unix` *und* `/permissive/edu/CA`
Apache:	`/permissive/web`
MIT:	`/permissive/edu/MA`
WTFPL:	`/permissive/funny/free`
JSON:	`/permissive/funny/unfree`

b) Welcher absolute Pfad führt ins Verzeichnis des dritten Dateisystems? Über welche relativen Pfade erreicht man von dort die Dateien mit Inhalt „Apache“ und „GPL“?

- `/permissive/funny` *wie in der Aufgabenstellung genannt*
- `../web`
- `../../political/left`

c) Welche relativen Pfade führen aus dem Verzeichnis `political` zu „Unlicense“, „MIT“ und „WTFPL“?

- `anti` *nur so, der zweite Pfad ist verdeckt*
- `../permissive/edu/MA`
- `../permissive/funny/free`

d) Was muss man tun, um auf „CC-0“ zuzugreifen?

Der einzige Verzeichniseintrag zu dieser Datei ist von einem Mount Point verdeckt. Man muss also zuerst das dritte, dann das zweite Dateisystem aushängen. Danach ist „CC-0“ als `/permissive/zero` erreichbar.

e) Zu welchen Dateien kann man Hard Links in `/permissive/edu/` anlegen?

Zu „BSD“, „Apache“ und „MIT“. Dass zwei der Dateien schon einen Eintrag in diesem Verzeichnis haben, spielt keine Rolle. Man kann weitere Einträge mit anderen Namen erzeugen. Die restlichen fünf Dateien liegen in anderen Dateisystemen, deshalb sind Hard Links dorthin nicht möglich.

Aufgabe K22: Synchronisation in Java (Recherche)

Die Java–Klasse `Cell` (siehe Aufgabenstellung) ähnelt dem Kernobjekt „Zelle“ aus früheren Aufgaben. Eine Zelle ist entweder leer oder enthält einen Integer–Wert. Die Methode `give` legt einen Wert in die Zelle. Enthält die Zelle bereits einen Wert, wird dieser überschrieben. Die Methode `take` entnimmt den aktuellen Wert aus der Zelle. Ist die Zelle gerade leer, fliegt eine Exception.

Die Implementierung von `Cell` ist noch nicht gegen gleichzeitige Aufrufe durch Java–Threads abgesichert. Das sollen Sie im Verlauf dieser Aufgabe ändern.

a) Was könnte schief gehen, wenn zwei Java–Threads die Methoden einer Zelle gleichzeitig aufrufen? Betrachten sie alle drei Kombinationen der beiden Methoden.

---

give + give: Nichts geht schief. Ein Wert überschreibt den anderen, aber das ist das erwartete Verhalten.

take + take: Ein Wert könnte doppelt entnommen werden.

give + take:

- Der frisch abgelegte Wert könnte von `take` zurückgegeben werden, aber trotzdem in der Zelle gespeichert bleiben.
- `take` könnte den vorher abgelegten Wert liefern und die Zelle als leer markieren, so dass der frisch abgelegte Wert verloren geht.

b) Recherchieren Sie die Bedeutung von `synchronized` in Java. Was ist der Unterschied zwischen `synchronized` an einer Methode und vor einem Codeblock?

Hier keine Erklärung von `synchronized`. Sie sollen selbst recherchieren, nicht in der Musterlösung nachlesen.

An einer nicht statischen Methode bezieht sich `synchronized` auf `this` und umfasst den gesamten Code der Methode. Vor einem Codeblock kann man ein anderes Objekt als Sperre nutzen.

c) Ändern Sie die Methoden in `Cell` so ab, dass mehrere Java–Threads die gleiche Zelle sicher verwenden können.

Der einfachste Weg ist es, sowohl `give` als auch `take` als `synchronized` zu deklarieren. Für die noch folgenden Änderungen muss man die zweite Variante einsetzen, wie unten gezeigt: `synchronized (guard)`

d) Recherchieren Sie die Methoden `wait()`, `notify()` und `notifyAll()` in Java. Wie hängen sie mit `synchronized` zusammen? Was sind *spurious wakeups*?

Hier keine Erklärung der Methoden.

Diese Methoden funktionieren nur innerhalb eines `synchronized`–Blocks für das Objekt, an dem man sie aufruft. `wait()` gibt die `synchronized`–Sperre frei, solange der Thread blockiert ist. Ein *spurious wakeup* lässt `wait()` zurückkehren, obwohl es noch nicht sollte.

e) Schreiben Sie `Cell` so um, dass `take()` auf einen Wert wartet, statt eine Exception zu werfen. Nutzen Sie das Objekt `guard` zur Synchronisation.

`take()` muss `guard.wait()` aufrufen, `give()` `guard.notify()`. Da nur einer der wartenden Threads den neuen Wert entnehmen kann, wäre `notifyAll()` nicht sinnvoll. Die `synchronized`–Sperre muss hier auf `guard` gesetzt werden.

```
public void give(int value)
{
  synchronized (guard) {
    stored_value = value;
    holds_value  = true;
    guard.notify();
  }
}

public int take()
  throws InterruptedException
{
  synchronized (guard) {
    while (!holds_value)
      guard.wait();

    holds_value = false;
    return stored_value;
  }
}
```

Wegen *spurious wakeups* muss man `wait()` in einer Schleife aufrufen, welche die Bedingung nach jedem Aufwachen prüft. Die `InterruptedException` von `wait()` schlägt auf `take()` durch. Die Warteschleife sollte diese Exception nicht verschlucken, da sie zum Unterbrechen von wartenden Threads dient.

f) Welchen Vorteil bringt die Synchronisation an `guard` statt `this`?

`guard` ist ein privates Objekt, an dem nur die Methoden in `Cell` synchronisieren können. An `this`, also der Zelle selbst, können die Synchronisationsmethoden von überall her aufgerufen werden. Das würde die Verwendung innerhalb von `Cell` stören. Zum Beispiel könnte `this.notify()` in `give()` einen Java–Thread wecken, der nicht in `take()`, sondern sonstwo blockiert ist.

Aufgabe K23: Übersetzungstabelle

Auf einem fiktiven Rechnersystem sind Adressen 20 Bit lang. Die Seitengröße liegt bei 4 KB, also 12 Bit. Eine fiktive ausführbare Programmdatei für dieses Rechnersystem definiert folgende Segmente:

Zweck	Anfang	Größe
Code	`0x10000`	`0x3000`
Konstanten	`0x13000`	`0x1000`
Statische Daten	`0x14000`	`0x2000`

Der Programmlader des Rechnersystems lädt die Segmente in der genannten Reihenfolge. Anschließend legt er folgende Bereiche an, ebenfalls in der genannten Reihenfolge:

Zweck	Anfang	Größe
Stapel	`0xee000`	`0x2000`
Halde	`0x20000`	`0x4000`

Die Seiten jedes Segments werden vom Anfang zum Ende hin angefordert. Die Speicherverwaltung liefert freie Seiten des Hauptspeichers in folgender Reihenfolge:

`53 54 6c 6a b7 91 55 76 a3 85 84 83 ab ba b3 9f` (*Hexzahlen*)

Erstellen Sie die Übersetzungstabelle für den Programmstart.
Flags: C = Code, D = Daten, L = nur Lesen, S = Schreiben und Lesen

Logisch	Physisch	Flags
`0x10`	`0x53`	`C L`
`0x11`	`0x54`	`C L`
`0x12`	`0x6c`	`C L`
`0x13`	`0x6a`	`D L`
`0x14`	`0xb7`	`D S`
`0x15`	`0x91`	`D S`
`0xee`	`0x55`	`D S`
`0xef`	`0x76`	`D S`
`0x20`	`0xa3`	`D S`
`0x21`	`0x85`	`D S`
`0x22`	`0x84`	`D S`
`0x23`	`0x83`	`D S`

Aufgabe K24: Übersetzungstabelle

Auf einem fiktiven Rechnersystem sind Adressen 20 Bit lang. Die Seitengröße liegt bei 4 KB, also 12 Bit. Eine fiktives Programm für dieses Rechnersystem verwendet eine dynamisch gebundene Bibliothek als Plugin. Die Programm- bzw. Bibliotheksdateien definieren folgende Segmente:

	Programm		Bibliothek	*Lader wählt*
Zweck	Anfang	Größe	Größe	Anfang
Code	`0x01000`	`0x2000`	`0x2000`	`0xe0000`
Konstanten	`0x04000`	`0x1000`	`0x1000`	`0xe3000`
Statische Daten	`0x06000`	`0x1000`	`0x2000`	`0xe5000`

Der Programm- und Bibliothekslader bearbeitet Segmente in der genannten Reihenfolge. Nach dem Laden des Programms legt er folgende Segmente an, wiederum in der genannten Reihenfolge. Nach dem Start veranlasst die Instanz das Laden des Plugins.

Zweck	Anfang	Größe
Halde	`0x10000`	`0x3000`
Stapel	`0xfd000`	`0x2000`

Die Seiten jedes Segments werden vom Anfang zum Ende hin angefordert. Die Speicherverwaltung liefert freie Seiten des Hauptspeichers in folgender Reihenfolge:

`89 b6 35 30 7a 42 c5 c0 dd d3 84 ab 85 50 53 62 a4` (*Hexzahlen*)

Erstellen Sie die Übersetzungstabelle nach dem Laden der Bibliothek.
Flags: C = Code, D = Daten, L = nur Lesen, S = Schreiben und Lesen

Logisch	Physisch	Flags
`0x01`	`0x89`	`C L`
`0x02`	`0xb6`	`C L`
`0x04`	`0x35`	`D L`
`0x06`	`0x30`	`D S`
`0x10`	`0x7a`	`D S`
`0x11`	`0x42`	`D S`
`0x12`	`0xc5`	`D S`
`0xfd`	`0xc0`	`D S`
`0xfe`	`0xdd`	`D S`
`0xe0`	`0xd3`	`C L`
`0xe1`	`0x84`	`C L`
`0xe3`	`0xab`	`D L`
`0xe5`	`0x85`	`D S`
`0xe6`	`0x50`	`D S`