Aufgabe V1: Segmente

Im Rahmen dieser Aufgabe definiert eine ausführbare Datei die drei Segmente Code, Konstanten und statische Daten. Das gleiche gilt für eine dynamisch gebundene Bibliothek. Eine Instanz im Speicher benötigt außerdem noch eine Halde und einen Stapel. Der Programmlader hält unveränderbare Segmente nur einmal im Speicher. Er stellt sie allen Instanzen zur Verfügung, die sie benötigen.

- `ProgP` verwendet `libL`
- `ProgQ` verwendet `libL` und `libM`
- `ProgR` verwendet `libM` und `libX`

Geben sie für die folgenden Instanzen an, wie viele Adressräume, Einblendungen und Segmente der Programmlader neu erzeugt. Die Instanzen aus den vorhergehenden Teilaufgaben befinden sich jeweils schon bzw. noch im Speicher.

a) Eine Instanz von `ProgP` startet.

Adressräume ______ Einblendungen ______ Segmente ______

b) Eine zweite Instanz von `ProgP` startet.

Adressräume ______ Einblendungen ______ Segmente ______

c) Eine Instanz von `ProgQ` startet.

Adressräume ______ Einblendungen ______ Segmente ______

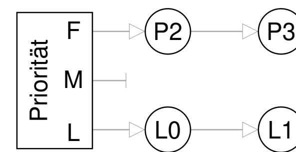
d) Eine Instanz von `ProgR` startet.

Adressräume ______ Einblendungen ______ Segmente ______

Aufgabe V2: Bereitmenge und Aufgreifstrategie

Ein fiktives Betriebssystem unterstützt die drei Prioritäten **F**lott, **M**oderat und **L**ungern. Die Leerlaufprozesse haben die niedrigste Priorität L, alle anderen Prozesse entweder F oder M. Prozesse mit gleicher Priorität sind nach dem Zeitpunkt geordnet, an dem sie in die Bereitmenge gelangen: Wer zuerst kommt, mahlt zuerst.
Dieses Betriebssystem arbeitet auf einem Rechner mit zwei Prozessoren (CPU0, CPU1). Es existieren vier normale Prozesse (P0–P3) mit flotter Priorität, die auf jedem Prozessor ablaufen können. Die beiden Leerlaufprozesse (L0, L1) sind dem jeweiligen Prozessor fest zugeordnet. Prozesse erhalten nur Rechenzeit, wenn kein Prozess mit einer höheren Priorität bereit steht. Zwischen Prozessen mit gleicher Priorität wird nach Ablauf einer Zeitscheibe umgeschaltet.
Die folgende Tabelle beschreibt den Systemzustand zu Beginn der Beobachtung. P0 und P1 rechnen auf CPU0 bzw. CPU1, in der Bereitmenge steht P2 vor P3. Unter der Tabelle ist eine chronologische Liste von Ereignissen aufgeführt, die die Prozessorzuteilung oder die Bereitmenge ändern können. Ereignisse wirken sich sofort auf alle Prozessoren aus.

CPU0	**CPU1**	**Bereitmenge**
P0	P1	{P2, P3; L0, L1}
...	...	...



1. Zeitscheibe CPU0 läuft ab
2. Zeitscheibe CPU1 läuft ab
3. P2 vermindert eigene Priorität
4. Zeitscheibe CPU1 läuft ab
5. Zeitscheibe CPU0 läuft ab
6. P1 vermindert Priorität von P3
7. P0 endet
8. Zeitscheibe CPU1 läuft ab
9. P1 erhöht Priorität von P3
10. P1 endet
11. P2 endet
12. P3 endet

a) Die Tabelle auf der folgenden Seite enthält eine Zeile pro Ereignis. Tragen Sie jeweils den Zustand nach den Änderungen ein, die sich durch das Ereignis ergeben. Prozesse mit moderater Priorität können Sie zum Beispiel durch ein Minuszeichen kenntlich machen ($P2^{-}$).

b) Bei welchen Ereignissen würde sich das System anders verhalten, wenn sie sich nicht sofort auf den anderen Prozessor auswirkten?

#	**CPU0**	**CPU1**	**Bereitmenge**
0	P0	P1	{P2, P3; L0, L1}
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			

Aufgabe V3: Warten

In einem System laufen vier Prozesse P0…P3, die über zwei Kernobjekte Za und Zb vom Typ „Zelle“ interagieren. Eine 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.
Jede Zelle verwaltet eine Wartemenge für Aufrufer von `take`, die noch keinen Wert erhalten haben. Ein Aufruf von `give` übergibt den Wert direkt an einen der Wartenden. In Za kommt eine Reihum-Strategie (FIFO, engl.: *first in, first out*) zum Einsatz. In Zb sind die wartenden Prozessen nach absteigenden Nummern geordnet. Der Prozess mit der höchsten Nummer erhält den nächsten übergebenen Wert. Die folgende Tabelle zeigt den Zustand der Zellen zu Beginn. Beide sind leer, ebenso wie ihre Wartemengen.

Zelle Za		**Zelle** Zb	
Wert	Wartemenge	Wert	Wartemenge
—	{}	—	{}

Füllen Sie die Tabelle auf der nächsten Seite. Für jeden Aufruf der folgenden Liste ist eine Zeile vorgesehen. Jede Zeile beschreibt den Systemzustand *nach* der jeweiligen Operation.

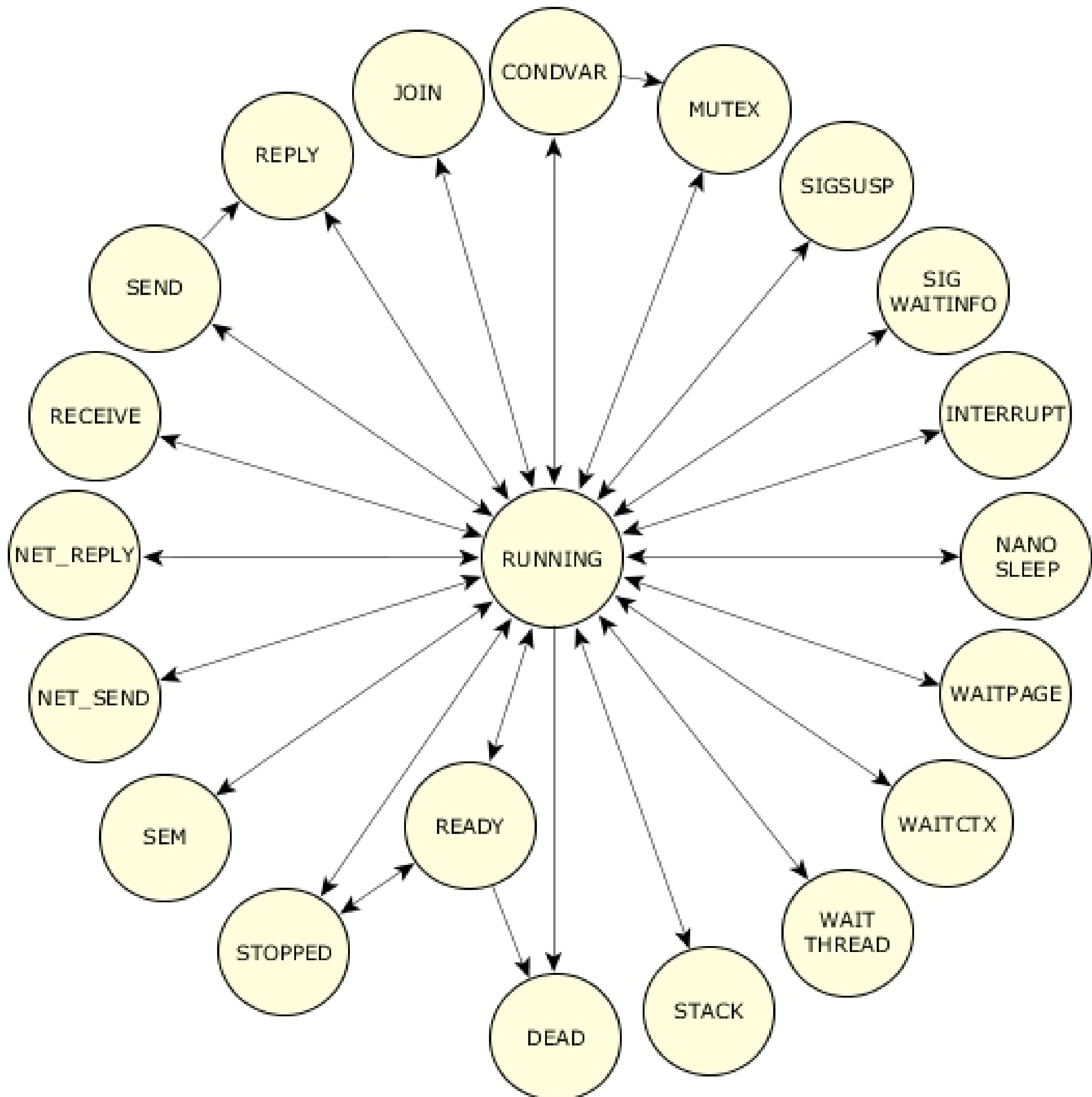
1. P0: $Za.give(42)$
2. P0: $Zb.take()$
3. P1: $Za.give(815)$
4. P1: $Zb.take()$
5. P2: $Zb.give(174)$
6. P2: $Za.take()$
7. P3: $Zb.give(321)$
8. P3: $Za.take()$
9. $P?_{174}$: $Zb.give(471)$
10. $P?_{174}$: $Za.take()$
11. $P?_{815}$: $Za.give(666)$
12. $P?_{815}$: $Za.take()$
13. $P?_{321}$: $Za.give(256)$
14. $P?_{321}$: $Zb.take()$
15. $P?_{666}$: $Za.give(768)$
16. $P?_{666}$: $Zb.take()$
17. $P?_{256}$: $Za.give(625)$
18. $P?_{256}$: $Zb.take()$
19. $P?_{768}$: $Zb.give(169)$
20. $P?_{768}$: $Za.take()$

$P?_{nnn}$ steht für denjenigen Prozess, der vorher bei einem `take` den Wert nnn erhalten hat. Sonst wäre es zu einfach, die Lösung aus der Aufgabenstellung abzulesen.

	Zelle Za		**Zelle** Zb	
#	Wert	Wartemenge	Wert	Wartemenge
0	—	{}	—	{}
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				
16				
17				
18				
19				
20				

Aufgabe V4: Prozesszustände

Der Kern des Betriebssystems QNX Neutrino unterscheidet 21 Prozesszustände. Ordnen Sie diese den Zuständen aus der Vorlesung zu. Beschreibungen aus der englischen Dokumentation von QNX Neutrino folgen auf der nächsten Seite.



Note that, in addition to the transitions shown above, a thread can move from any state (except DEAD) to READY.

Quelle: QNX® Software Development Platform 6.6, *Abschnitt:* Thread life cycle
`http://www.qnx.com/developers/docs/660/topic/index.jsp`
`.../docs/660/topic/com.qnx.doc.neutrino.sys_arch/topic/kernel_Life_Cycle.html`

CONDVAR: The thread is blocked on a condition variable
(e.g., it called `pthread_cond_wait()`).

DEAD: The thread has terminated and is waiting for a join by another thread.

INTERRUPT: The thread is blocked waiting for an interrupt
(i.e., it called `InterruptWait()`).

JOIN: The thread is blocked waiting to join another thread
(e.g., it called `pthread_join()`).

MUTEX: The thread is blocked on a mutual exclusion lock
(e.g., it called `pthread_mutex_lock()`).

NANOSLEEP: The thread is sleeping for a short time interval
(e.g., it called `nanosleep()`).

NET_REPLY: The thread is waiting for a reply to be delivered across the network (i.e., it called `MsgReply*()`).

NET_SEND: The thread is waiting for a pulse or signal to be delivered across the network (i.e., it called `MsgSendPulse()`, `MsgDeliverEvent()`, or `SignalKill()`).

READY: The thread is waiting to be executed while the processor executes another thread of equal or higher priority.

RECEIVE: The thread is blocked on a message receive (e.g., it called `MsgReceive()`).

REPLY: The thread is blocked on a message reply (i.e., it called `MsgSend()`, and the server received the message).

RUNNING: The thread is being executed by a processor. The kernel uses an array (with one entry per processor in the system) to keep track of the running threads.

SEM: The thread is waiting for a semaphore to be posted
(i.e., it called `SyncSemWait()`).

SEND: The thread is blocked on a message send (e.g., it called `MsgSend()`, but the server hasn't yet received the message).

SIGSUSPEND: The thread is blocked waiting for a signal
(i.e., it called `sigsuspend()`).

SIGWAITINFO: The thread is blocked waiting for a signal
(i.e., it called `sigwaitinfo()`).

STACK: The thread is waiting for the virtual address space to be allocated for the thread's stack (parent will have called `ThreadCreate()`).

STOPPED: The thread is blocked waiting for a SIGCONT signal.

WAITCTX: The thread is waiting for a noninteger (e.g., floating point) context to become available for use.

WAITPAGE: The thread is waiting for physical memory to be allocated for a virtual address.

WAITTHREAD: The thread is waiting for a child thread to finish creating itself (i.e., it called `ThreadCreate()`).

Aufgabe V5: Kritische Abschnitte

Die Java–Klasse auf der nächsten Seite implementiert einen Puffer mittels eines Arrays und eines Zählers. Der Puffer arbeitet nach dem LIFO–Prinzip (engl.: *Last In, First Out*), wie ein Stapel. Er steht stellvertretend für beliebige, nicht–triviale Datenstrukturen. Die Implementierung ist nicht gegen gleichzeitige Aufrufe durch mehrere Prozesse abgesichert.
Im Folgenden rufen zwei Prozesse, d.h. Java–Threads, gleichzeitig Methoden am selben Puffer auf. Der Puffer enthält bereits einige Werte, aber auch noch freie Plätze. Die Operationen stoßen also weder an den oberen noch an den unteren Rand des Arrays. Gehen Sie davon aus, dass zwischen zwei Zeilen des Quelltexts jeweils der andere Prozess zum Zuge kommen könnte.
Wenn beide Prozesse `fetch` aufrufen wäre das korrekte Verhalten, dass einer den obersten Wert vom Stapel nimmt, der andere den zweitobersten. Das kann aber schiefgehen. Bei folgender Reihenfolge der Befehle entnehmen beide Prozesse den obersten Wert:

Prozess A: `fetch`	**Prozess B**: `fetch`
$\text{which}_A = \text{items-1}$	
	$\text{which}_B = \text{items-1}$
$\text{result}_A = \text{buffer[which}_A]$	
	$\text{result}_B = \text{buffer[which}_B]$
$\text{items} = \text{which}_A$	
	$\text{items} = \text{which}_B$
return result_A	
	return result_B

Lokale Variablen wie `which` und `result` existieren pro Prozess, deshalb die Indizes $_A$ und $_B$. Im Quelltext sind sie als `final` deklariert. Die Attribute `items` und `buffer` des Puffers sind für beide Prozesse die selben.

a) Welche Zeilen muss Prozess A am Stück ausführen, damit Prozess B nicht mehr den gleichen Wert vom Stapel nehmen kann?

   Diese Anweisungen bilden einen *kritischen Abschnitt* in `fetch`.

b) Beide Prozesse rufen `store` auf. Was ist das korrekte Verhalten? Was könnte wie schiefgehen? Welche Anweisungen bilden den kritischen Abschnitt in `store`?

c) Prozess A ruft `fetch`, Prozess B `store`. Welches sind die zwei korrekten Verhalten bei dieser Kombination? Auf welche zwei Arten könnte es schiefgehen? Gibt es neue kritische Abschnitte für die Kombination?

d) Prozess A ruft `store`, Prozess B `toString`. Welches sind die zwei korrekten Verhalten bei dieser Kombination? Was könnte wie schiefgehen? Welche Anweisungen bilden den kritischen Abschnitt in `toString`?

```java
import java.lang.reflect.Array;


public class BufferLIFO<T>
{
  protected final T[] buffer;
  protected       int items;

  @SuppressWarnings("unchecked")
  public BufferLIFO(Class<T> clazz, int capacity)
   {
     buffer = // new T[capacity]
       (T[]) Array.newInstance(clazz, capacity);
     items  = 0;
   }

  public void store(T item)
   {
     final int which = items;
     buffer[which] = item;
     items = which+1;
   }

  public T fetch()
   {
     final int which = items-1;
     final T result = buffer[which];
     items = which;
     return result;
   }

  public String toString()
   {
     StringBuilder sb = new StringBuilder(80);
     sb.append(getClass().getName())
       .append('[').append(items)
       .append('/').append(buffer.length)
       .append(']');
     return sb.toString();
   }
}
```

Aufgabe V6: Semaphore im Einsatz

Eine Pufferverwaltung auf Anwendungsebene erlaubt es, Puffer fester Größe anzufordern und wieder freizugeben. Die Anzahl der Puffer wird einmalig bei der Initialisierung festgelegt. Die Puffer sind ab 0 durchnummeriert. Die Verwaltung verwendet einen Bitvektor entsprechender Länge, in dem 0 einen freien und 1 einen belegten Puffer anzeigt. Die Operationen laufen wie folgt ab:

Anfordern: *kein Argument, Rückgabewert ist eine Puffernummer oder ein Fehler*

1. im Bitvektor nach einer 0 suchen
2. nichts gefunden ⇒ Fehler zurückgeben
3. gefundenes Bit auf 1 setzen
4. entsprechende Puffernummer zurückgeben

Freigeben: *Argument ist eine Puffernummer, kein Rückgabewert*

8. im Bitvektor das entsprechende Bit auf 0 setzen

Die Pufferverwaltung ist noch nicht für parallele Aufrufe durch mehrere Prozesse (engl.: *threads*) gleichzeitig ausgelegt. Gehen Sie davon aus, dass die Anwendung nur gültige Aufrufe durchführt, also nur zuvor angeforderte Puffer freigibt.

a) Bestimmen Sie die kritischen Abschnitte beider Operationen.

b) Sichern Sie die Pufferverwaltung mit einem Semaphor `M` so ab, dass höchstens ein kritischer Abschnitt gleichzeitig ablaufen kann. Mit welchem Wert wird `M` initialisiert? Wo müssen Aufrufe von `M.P()` bzw. `M.V()` stattfinden?

c) Ändern Sie die Pufferverwaltung mit einem zweiten Semaphor `Z` so, dass das Anfordern auf einen freien Puffer wartet, statt einen Fehler zurückzugeben. Mit welchem Wert wird `Z` initialisiert? Wo müssen Aufrufe von `Z.P()` bzw. `Z.V()` stattfinden?

d) Was kann schiefgehen, wenn zwei Prozesse gleichzeitig die ungesicherte Pufferverwaltung nutzen, also ohne das Semaphor `M`?

- beide rufen **Anfordern**
- beide rufen **Freigeben**
- einer ruft **Anfordern**, der andere **Freigeben**

Aufgabe V7: Semaphore für Erzeuger/Verbraucher

Eine Pufferverwaltung zur Datenflusskontrolle unterscheidet zwischen leeren und gefüllten Puffern. Alle Puffer haben die gleiche Größe. Zwei Arten von Prozessen nutzen diese Verwaltung: *Erzeuger* fordern leere Puffer an und geben gefüllte zurück, *Verbraucher* fordern gefüllte Puffer an und geben leere zurück.
Die Puffer sind ab 0 durchnummeriert, ihre Gesamtzahl wird einmalig bei der Initialisierung festgelegt. Die Verwaltung verwendet zwei Bitvektoren entsprechender Länge. Der Bitvektor RB gibt für jeden Puffer an, ob dieser in Ruhe (0) oder in Bearbeitung (1) ist. Der Bitvektor LG gibt für jeden Puffer in Ruhe an, ob er leer (0) oder gefüllt (1) ist. Die Operationen laufen wie folgt ab:

Leer Anfordern: *kein Argument, Rückgabewert ist eine Puffernummer*

1. einen Puffer suchen, für den RB und LG auf 0 stehen
2. entsprechendes Bit in RB auf 1 setzen
3. entsprechende Puffernummer zurückgeben

Voll Zurückgeben: *Argument ist eine Puffernummer, kein Rückgabewert*

1. entsprechende Bits in RB auf 0 und in LG auf 1 setzen

Voll Anfordern: *kein Argument, Rückgabewert ist eine Puffernummer*

1. einen Puffer suchen, für den RB auf 0 und LG auf 1 steht
2. entsprechendes Bit in RB auf 1 setzen
3. entsprechende Puffernummer zurückgeben

Leer Zurückgeben: *Argument ist eine Puffernummer, kein Rückgabewert*

1. entsprechende Bits in RB und LG auf 0 setzen

In diesen Abläufen fehlt noch die Synchronisation der Prozesse. Gehen Sie davon aus, dass diese nur gültige Aufrufe durchführen, also nur zuvor angeforderte Puffer zurückgeben. Das Anfordern wartet jeweils, bis ein entsprechender Puffer verfügbar ist.

a) Die Verwaltung nutzt drei Semaphore `M`, `L` und `G`. Mit welchen Werten werden diese initialisiert? Wo finden die Aufrufe von `P()` bzw. `V()` statt?

b) Ist sichergestellt, dass jeder gefüllte Puffer auch wieder geleert wird, sofern Erzeuger und Verbraucher ihren Aufgaben nachkommen? Falls nicht, was muss man ändern?

c) In Fehlersituationen kann es notwendig sein, dass Erzeuger einen leeren Puffer zurückgeben, oder Verbraucher einen gefüllten. Funktioniert das mit den beschriebenen Abläufen und der Synchronisation? Falls nicht, was muss man ändern?

Aufgabe V8: Kanal mit Puffer

Ein Kanal ist ein Treffpunkt für Prozesse, die Nachrichten senden oder empfangen. Trifft ein Sender auf einen Empfänger, überträgt der Kanal die Nachricht. Für beide Operationen bietet der Kanal verschiedene Kopplungsformen an. Er ist als Kernobjekt implementiert und verwendet folgende Attribute:

- Menge **NP** für Tupel (Nachrichtenreferenz, Prozess). Initial leer.
  Der Prozess darf `null` sein.
- Menge **ZP** für Tupel (Zielreferenz, Prozess). Initial leer.
  Der Prozess darf nicht `null` sein.
- Pufferspeicher **PS** für mehrere Nachrichten, Größe begrenzt. Initial leer.
  Man kann den freien Platz abfragen, Nachrichten hinein- und herauskopieren, und sie einzeln aus dem Pufferspeicher löschen.

Ungültige Fälle wären:

1. Sowohl **NP** als auch **ZP** enthalten ein Element.
2. **PS** enthält eine Nachricht, welche **NP** nicht referenziert.

Es gibt eine maximale Nachrichtengröße für jeden Kanal. Bei allen Sendeoperationen übergibt der Aufrufer eine Referenz auf die Nachricht in seinem Adressraum. Bei allen Empfangsoperationen übergibt der Aufrufer eine Referenz auf einen Zielbereich, d.h. einen Empfangspuffer, in seinem Adressraum. Sie können für diese Aufgabe davon ausgehen, dass Empfangspuffer mindestens so groß sind wie die maximale Nachrichtengröße.

Die Abläufe des synchronen Sendens (`sendSyn`) mit Referenzablage und des versuchenden Empfangs (`receiveTry`) sind auf der folgenden Seite beschrieben.

a) Beschreiben Sie den Ablauf des synchronen Empfangens. (`receiveSyn`)

b) Beschreiben Sie den Ablauf des asynchronen Sendens (`sendAsyn`) mit Wertablage. Falls nicht genug Pufferspeicher frei ist, scheitert die Operation.

c) Was bedeuten die beiden ungültigen Fälle?

d) Die Operation `receiveSyn` wird von n Prozessen in Dauerschleife aufgerufen. Schätzen Sie den Speicherbedarf des Kanals dafür ab.

e) Die Operation `sendAsyn` wird von m Prozessen in Dauerschleife aufgerufen. Schätzen Sie den Speicherbedarf des Kanals dafür ab.

Das synchrone Senden (`sendSyn`) mit Referenzablage in V8 läuft wie folgt ab:

- Nachricht zu groß? ⇒ Abbruch mit Fehler
- **ZP** enthält Element?

ja: Tupel (Zielreferenz, Prozess) aus **ZP** nehmen
Nachricht in Zielbereich kopieren
Prozess deblockieren

nein: Aufrufer blockieren
(Nachrichtenreferenz, Aufrufer) in **NP** legen

Das versuchende Empfangen (`receiveTry`) in V8 läuft wie folgt ab:

- **NP** enthält Element?

ja: Tupel (Nachrichtenref., Prozess) aus **NP** nehmen
Nachricht in Zielbereich kopieren
Prozess deblockieren, falls nicht `null`
Nachricht aus **PS** löschen, falls von dort
Erfolg melden

nein: Misserfolg melden

Aufgabe V9: Adressübersetzung

In dieser Aufgabe sind Adressen 20 Bit lang, bei der Übersetzung bleiben die letzten 12 Bit unverändert. Gegeben sind die folgenden drei Übersetzungstabellen (ÜT). Alle Tabelleneinträge und alle Adressen in den Teilaufgaben sind als Hexzahlen angegeben.

ÜT A

Logisch	Physisch
40	82
41	83
42	86
20	81
21	84
bb	8f
cc	cc

ÜT B

Logisch	Physisch
40	88
41	8b
42	85
20	81
21	80
bb	88
cc	8f
ee	8d

ÜT C

Logisch	Physisch
40	8c
41	89
20	81
21	84
bb	8d
ee	8f

a) Übersetzen Sie die folgenden logischen Adressen mit jeder der Tabellen.
`40004`, `41010`, `4220a`, `44531`, `2038d`, `21f08`, `bb0b1`, `ccafe`, `eed73`

b) Geben Sie für jede der Tabellen an, mit welchen logischen Adressen man die folgenden physischen Adressen erreicht.
`82000`, `85222`, `89111`, `81358`, `84248`, `8fd7b`, `88888`, `ccccc`