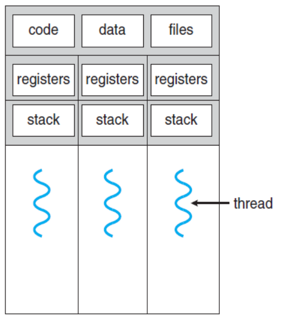
# Systemprogrammierung Beispiel 1:



- ## Producer-consumer problem

In multithreaded programs there is often a division of labor between threads. In one common pattern, some threads are producers and some are consumers. Producers create items of some kind and add them to a data structure; consumers remove the items and process them.

Event-driven programs are a good example. An "event" is something that happens that requires the program to respond: the user presses a key or moves the mouse, a block of data arrives from the disk, a packet arrives from the network, a pending operation completes.

Whenever an event occurs, a producer thread creates an event object and adds it to the event buffer. Concurrently, consumer threads take events out of the buffer and process them. In this case, the consumers are called "event handlers."

There are several synchronization constraints that we need to enforce to make this system work correctly:

- While an item is being added to or removed from the buffer, the buffer is in an inconsistent state. Therefore, threads must have exclusive access to the buffer.

- If a consumer thread arrives while the buffer is empty, it blocks until a producer adds a new item.

Assume that producers perform the following operations over and over:

Basic producer code

```
event = waitForEvent()
buffer.add(event)
```

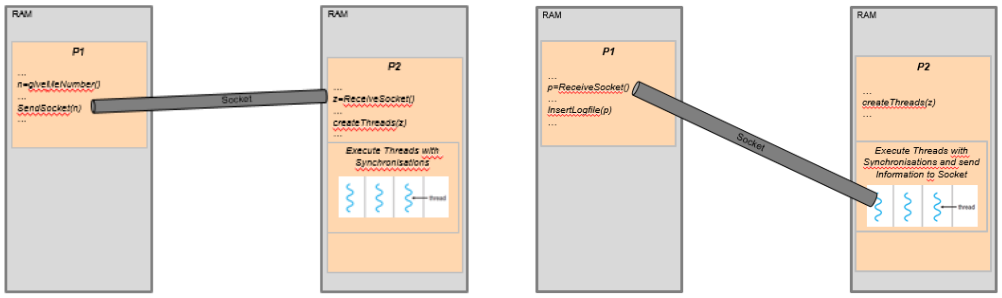
Also, assume that consumers perform the following operations:

Basic consumer code

```
event = buffer.get()
event.process()
```

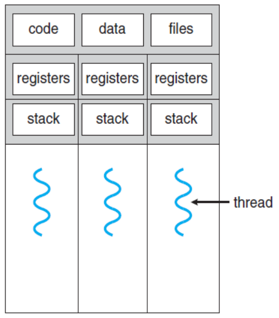
- **Aufgaben:**
  - **Synchronisation**
    Analysieren Sie das obige Problem und schreiben sie ein entsprechendes C-Programm, welches die Aufgabestellung realisiert und löst. Wählen sie passende Programmkonstrukte um parallele Strukturen, Ein- und Ausgaben und Synchronisationsmechanismen zu realisieren.

  - **Sockets**
    Erweitern Sie die obige Aufgabenstellung mit einer Socketanwendung. Mit dieser Erweiterung können z.B. Eingaben übergeben werden oder Ausgaben an einen weiteren Serverprozess übertragen werden.



- **Abgabe:**
  Abzugeben sind ein dokumentierter SourceCode und ein Funktionsnachweis in Form von ScreenShots, Unix-Befehlen, Programmausgaben, …
  Abzugeben ist auch eine prinzipielle Beschreibung des Problems.

## Systemprogrammierung Beispiel 2:



- ### Dining philosophers problem

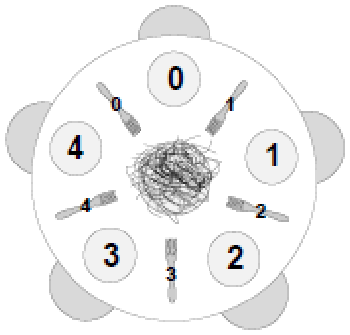
The Dining Philosophers Problem was proposed by Dijkstra in 1965, when dinosaurs ruled the earth [3]. It appears in a number of variations, but the standard features are a table with five plates, five forks (or chopsticks) and a big bowl of spaghetti. Five philosophers, who represent interacting threads, come to the table and execute the following loop:

Basic philosopher loop

```
while True:
    think()
    get_forks()
    eat()
    put_forks()
```

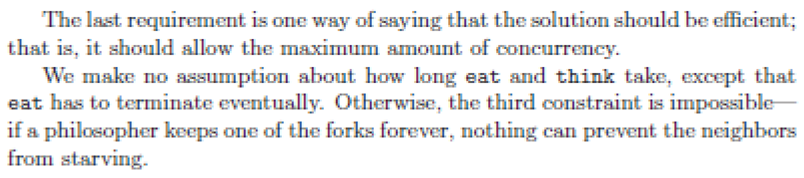
The forks represent resources that the threads have to hold exclusively in order to make progress. The thing that makes the problem interesting, unrealistic, and unsanitary, is that the philosophers need *two* forks to eat, so a hungry philosopher might have to wait for a neighbor to put down a fork.

Assume that the philosophers have a local variable `i` that identifies each philosopher with a value in (0..4). Similarly, the forks are numbered from 0 to 4, so that Philosopher $i$ has fork $i$ on the right and fork $i+1$ on the left. Here is a diagram of the situation:



Assuming that the philosophers know how to `think` and `eat`, our job is to write a version of `get_forks` and `put_forks` that satisfies the following constraints:

- Only one philosopher can hold a fork at a time.
- It must be impossible for a deadlock to occur.
- It must be impossible for a philosopher to starve waiting for a fork.
- It must be possible for more than one philosopher to eat at the same time.

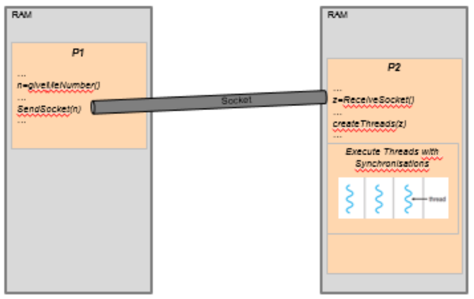


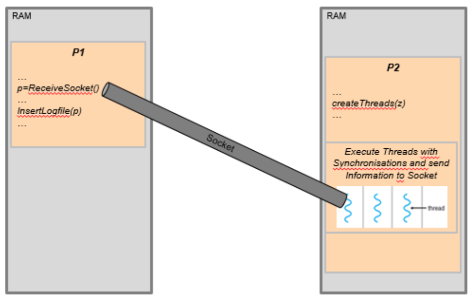
The last requirement is one way of saying that the solution should be efficient; that is, it should allow the maximum amount of concurrency.

We make no assumption about how long `eat` and `think` take, except that `eat` has to terminate eventually. Otherwise, the third constraint is impossible—if a philosopher keeps one of the forks forever, nothing can prevent the neighbors from starving.

- **Aufgaben:**
  - **Synchronisation**
    Analysieren Sie das obige Problem und schreiben sie ein entsprechendes C-Programm, welches die Aufgabestellung realisiert und löst. Wählen sie passende Programmkonstrukte um parallele Strukturen, Ein- und Ausgaben und Synchronisationsmechanismen zu realisieren.

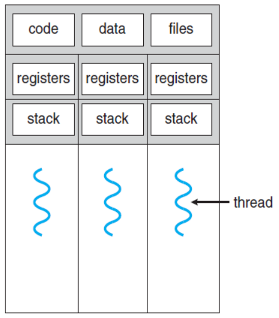
  - **Sockets**
    Erweitern Sie die obige Aufgabenstellung mit einer Socketanwendung. Mit dieser Erweiterung können z.B. Eingaben übergeben werden oder Ausgaben an einen weiteren Serverprozess übertragen werden.





- **Abgabe:**
  Abzugeben sind ein dokumentierter SourceCode und ein Funktionsnachweis in Form von ScreenShots, Unix-Befehlen, Programmausgaben, …
  Abzugeben ist auch eine prinzipielle Beschreibung des Problems.

# Systemprogrammierung Beispiel 3:



- **Cigarette smokers problem**

The cigarette smokers problem problem was originally presented by Suhas Patil [8], who claimed that it cannot be solved with semaphores. That claim comes with some qualifications, but in any case the problem is interesting and challenging.

Four threads are involved: an agent and three smokers. The smokers loop forever, first waiting for ingredients, then making and smoking cigarettes. The ingredients are tobacco, paper, and matches.

We assume that the agent has an infinite supply of all three ingredients, and each smoker has an infinite supply of one of the ingredients; that is, one smoker has matches, another has paper, and the third has tobacco.

The agent repeatedly chooses two different ingredients at random and makes them available to the smokers. Depending on which ingredients are chosen, the smoker with the complementary ingredient should pick up both resources and proceed.

For example, if the agent puts out tobacco and paper, the smoker with the matches should pick up both ingredients, make a cigarette, and then signal the agent.

To explain the premise, the agent represents an operating system that allocates resources, and the smokers represent applications that need resources. The problem is to make sure that if resources are available that would allow one more applications to proceed, those applications should be woken up. Conversely, we want to avoid waking an application if it cannot proceed.

Based on this premise, there are three versions of this problem that often appear in textbooks:

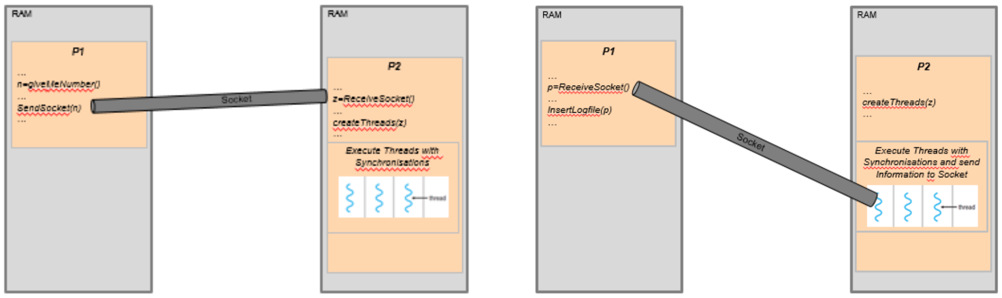
**The impossible version:** Patil's version imposes restrictions on the solution. First, you are not allowed to modify the agent code. If the agent represents an operating system, it makes sense to assume that you don't want to modify it every time a new application comes along. The second restriction is that you can't use conditional statements or an array of semaphores. With these constraints, the problem cannot be solved, but as Parnas points out, the second restriction is pretty artificial [7]. With constraints like these, a lot of problems become unsolvable.

**The interesting version:** This version keeps the first restriction—you can't change the agent code—but it drops the others.

**The trivial version:** In some textbooks, the problem specifies that the agent should signal the smoker that should go next, according to the ingredients that are available. This version of the problem is uninteresting because it makes the whole premise, the ingredients and the cigarettes, irrelevant. Also, as a practical matter, it is probably not a good idea to require the agent to know about the other threads and what they are waiting for. Finally, this version of the problem is just too easy.

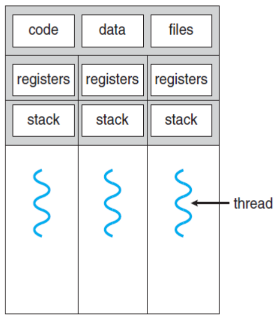
- ## Aufgaben:
  - **Synchronisation**
    Analysieren Sie das obige Problem und schreiben sie ein entsprechendes C-Programm, welches die Aufgabestellung realisiert und löst. Wählen sie passende Programmkonstrukte um parallele Strukturen, Ein- und Ausgaben und Synchronisationsmechanismen zu realisieren.

  - **Sockets**
    Erweitern Sie die obige Aufgabenstellung mit einer Socketanwendung. Mit dieser Erweiterung können z.B. Eingaben übergeben werden oder Ausgaben an einen weiteren Serverprozess übertragen werden.



- ## Abgabe:
  Abzugeben sind ein dokumentierter SourceCode und ein Funktionsnachweis in Form von ScreenShots, Unix-Befehlen, Programmausgaben, …
  Abzugeben ist auch eine prinzipielle Beschreibung des Problems.

## Systemprogrammierung Beispiel 4:



- **Readers-writers problem**

The next classical problem, called the Reader-Writer Problem, pertains to any situation where a data structure, database, or file system is read and modified by concurrent threads. While the data structure is being written or modified it is often necessary to bar other threads from reading, in order to prevent a reader from interrupting a modification in progress and reading inconsistent or invalid data.

As in the producer-consumer problem, the solution is asymmetric. Readers and writers execute different code before entering the critical section. The synchronization constraints are:

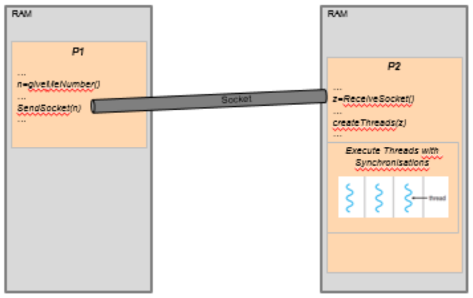
1. Any number of readers can be in the critical section simultaneously.
2. Writers must have exclusive access to the critical section.

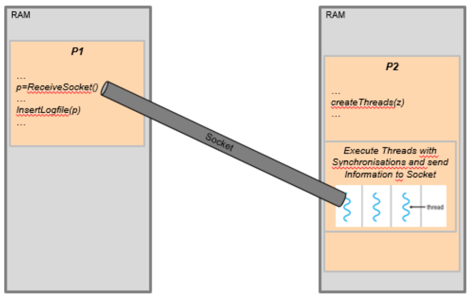
In other words, a writer cannot enter the critical section while any other thread (reader or writer) is there, and while the writer is there, no other thread may enter.

The exclusion pattern here might be called **categorical mutual exclusion**. A thread in the critical section does not necessarily exclude other threads, but the presence of one category in the critical section excludes other categories.

- ## Aufgaben:
  - **Synchronisation**
    Analysieren Sie das obige Problem und schreiben sie ein entsprechendes C-Programm, welches die Aufgabestellung realisiert und löst. Wählen sie passende Programmkonstrukte um parallele Strukturen, Ein- und Ausgaben und Synchronisationsmechanismen zu realisieren.

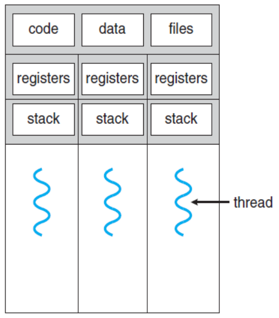
  - **Sockets**
    Erweitern Sie die obige Aufgabenstellung mit einer Socketanwendung. Mit dieser Erweiterung können z.B. Eingaben übergeben werden oder Ausgaben an einen weiteren Serverprozess übertragen werden.





- ## Abgabe:
  Abzugeben sind ein dokumentierter SourceCode und ein Funktionsnachweis in Form von ScreenShots, Unix-Befehlen, Programmausgaben, …
  Abzugeben ist auch eine prinzipielle Beschreibung des Problems.

# Systemprogrammierung Beispiel 5:



- ## Producer-consumer problem

In multithreaded programs there is often a division of labor between threads. In one common pattern, some threads are producers and some are consumers. Producers create items of some kind and add them to a data structure; consumers remove the items and process them.

Event-driven programs are a good example. An "event" is something that happens that requires the program to respond: the user presses a key or moves the mouse, a block of data arrives from the disk, a packet arrives from the network, a pending operation completes.

Whenever an event occurs, a producer thread creates an event object and adds it to the event buffer. Concurrently, consumer threads take events out of the buffer and process them. In this case, the consumers are called "event handlers."

There are several synchronization constraints that we need to enforce to make this system work correctly:

- While an item is being added to or removed from the buffer, the buffer is in an inconsistent state. Therefore, threads must have exclusive access to the buffer.
- If a consumer thread arrives while the buffer is empty, it blocks until a producer adds a new item.

Assume that producers perform the following operations over and over:

Basic producer code

```
event = waitForEvent()
buffer.add(event)
```

Also, assume that consumers perform the following operations:

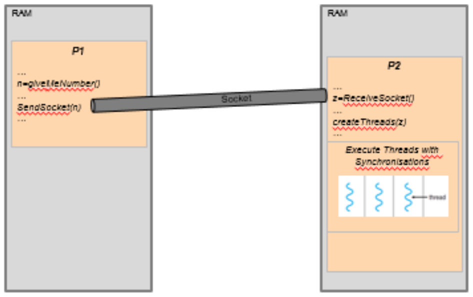
Basic consumer code

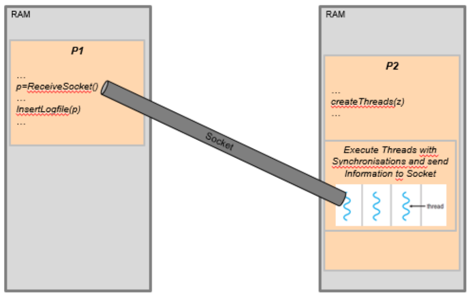
```
event = buffer.get()
event.process()
```

As specified above, access to the buffer has to be exclusive, but `waitForEvent` and `event.process` can run concurrently.

- ## Aufgaben:
  - **Synchronisation**
    Analysieren Sie das obige Problem und schreiben sie ein entsprechendes C-Programm, welches die Aufgabestellung realisiert und löst. Wählen sie passende Programmkonstrukte um parallele Strukturen, Ein- und Ausgaben und Synchronisationsmechanismen zu realisieren.

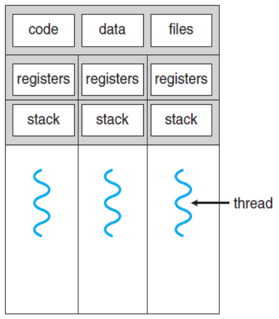
  - **Sockets**
    Erweitern Sie die obige Aufgabenstellung mit einer Socketanwendung. Mit dieser Erweiterung können z.B. Eingaben übergeben werden oder Ausgaben an einen weiteren Serverprozess übertragen werden.





- ## Abgabe:
  Abzugeben sind ein dokumentierter SourceCode und ein Funktionsnachweis in Form von ScreenShots, Unix-Befehlen, Programmausgaben, …
  Abzugeben ist auch eine prinzipielle Beschreibung des Problems.

# Systemprogrammierung Beispiel 6:



- **The Barbershop problem**

The original barbershop problem was proposed by Dijkstra. A variation of it appears in Silberschatz and Galvin's *Operating Systems Concepts* [10].

> A barbershop consists of a waiting room with $n$ chairs, and the barber room containing the barber chair. If there are no customers to be served, the barber goes to sleep. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy, but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber. Write a program to coordinate the barber and the customers.

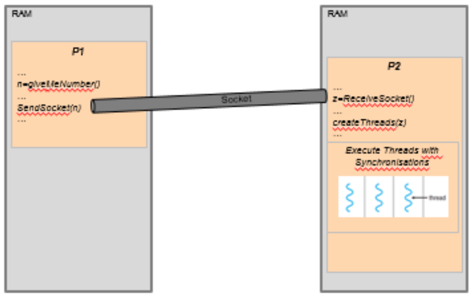
To make the problem a little more concrete, I added the following information:

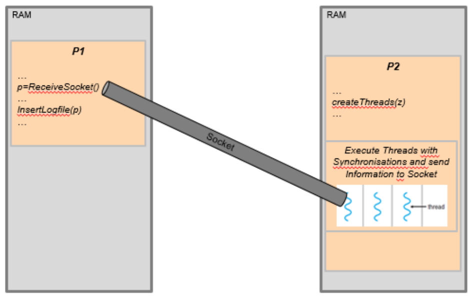
- Customer threads should invoke a function named `getHairCut`.
- If a customer thread arrives when the shop is full, it can invoke `balk`, which does not return.
- The barber thread should invoke `cutHair`.
- When the barber invokes `cutHair` there should be exactly one thread invoking `getHairCut` concurrently.

Write a solution that guarantees these constraints.

- ## Aufgaben:
  - **Synchronisation**
    Analysieren Sie das obige Problem und schreiben sie ein entsprechendes C-Programm, welches die Aufgabestellung realisiert und löst. Wählen sie passende Programmkonstrukte um parallele Strukturen, Ein- und Ausgaben und Synchronisationsmechanismen zu realisieren.

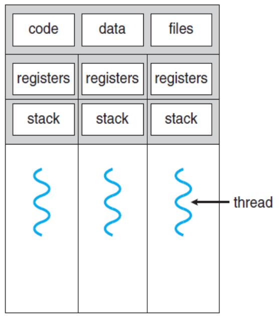
  - **Sockets**
    Erweitern Sie die obige Aufgabenstellung mit einer Socketanwendung. Mit dieser Erweiterung können z.B. Eingaben übergeben werden oder Ausgaben an einen weiteren Serverprozess übertragen werden.





- ## Abgabe:
  Abzugeben sind ein dokumentierter SourceCode und ein Funktionsnachweis in Form von ScreenShots, Unix-Befehlen, Programmausgaben, …
  Abzugeben ist auch eine prinzipielle Beschreibung des Problems.

# Systemprogrammierung Beispiel 7:



- **The Santa Claus problem**

This problem is from William Stallings's *Operating Systems* [11], but he attributes it to John Trono of St. Michael's College in Vermont.

> Stand Claus sleeps in his shop at the North Pole and can only be awakened by either (1) all nine reindeer being back from their vacation in the South Pacific, or (2) some of the elves having difficulty making toys; to allow Santa to get some sleep, the elves can only wake him when three of them have problems. When three elves are having their problems solved, any other elves wishing to visit Santa must wait for those elves to return. If Santa wakes up to find three elves waiting at his shop's door, along with the last reindeer having come back from the tropics, Santa has decided that the elves can wait until after Christmas, because it is more important to get his sleigh ready. (It is assumed that the reindeer do not want to leave the tropics, and therefore they stay there until the last possible moment.) The last reindeer to arrive must get Santa while the others wait in a warming hut before being harnessed to the sleigh.

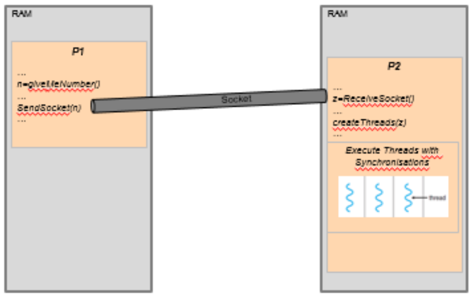
Here are some addition specifications:

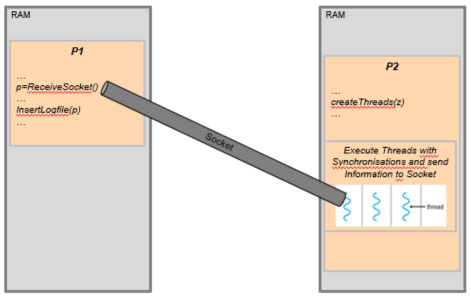
- After the ninth reindeer arrives, Santa must invoke `prepareSleigh`, and then all nine reindeer must invoke `getHitched`.
- After the third elf arrives, Santa must invoke `helpElves`. Concurrently, all three elves should invoke `getHelp`.
- All three elves must invoke `getHelp` before any additional elves enter (increment the elf counter).

Santa should run in a loop so he can help many sets of elves. We can assume that there are exactly 9 reindeer, but there may be any number of elves.

- **Aufgaben:**
  - **Synchronisation**
    Analysieren Sie das obige Problem und schreiben sie ein entsprechendes C-Programm, welches die Aufgabestellung realisiert und löst. Wählen sie passende Programmkonstrukte um parallele Strukturen, Ein- und Ausgaben und Synchronisationsmechanismen zu realisieren.

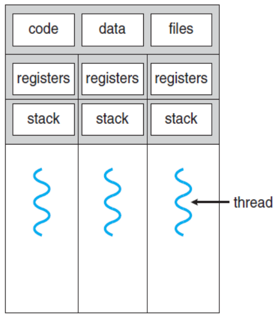
  - **Sockets**
    Erweitern Sie die obige Aufgabenstellung mit einer Socketanwendung. Mit dieser Erweiterung können z.B. Eingaben übergeben werden oder Ausgaben an einen weiteren Serverprozess übertragen werden.





- **Abgabe:**
  Abzugeben sind ein dokumentierter SourceCode und ein Funktionsnachweis in Form von ScreenShots, Unix-Befehlen, Programmausgaben, …
  Abzugeben ist auch eine prinzipielle Beschreibung des Problems.

# Systemprogrammierung Beispiel 8:



- **Building H2O problem**

This problem has been a staple of the Operating Systems class at U.C. Berkeley for at least a decade. It seems to be based on an exercise in Andrews's *Concurrent Programming* [1].

There are two kinds of threads, oxygen and hydrogen. In order to assemble these threads into water molecules, we have to create a barrier that makes each thread wait until a complete molecule is ready to proceed.

As each thread passes the barrier, it should invoke `bond`. You must guarantee that all the threads from one molecule invoke `bond` before any of the threads from the next molecule do.

In other words:

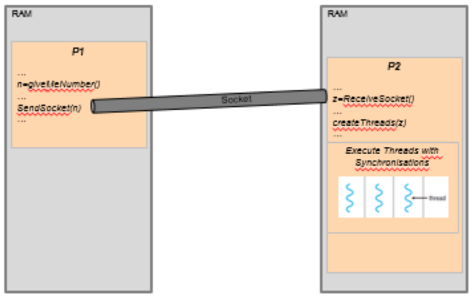
- If an oxygen thread arrives at the barrier when no hydrogen threads are present, it has to wait for two hydrogen threads.
- If a hydrogen thread arrives at the barrier when no other threads are present, it has to wait for an oxygen thread and another hydrogen thread.

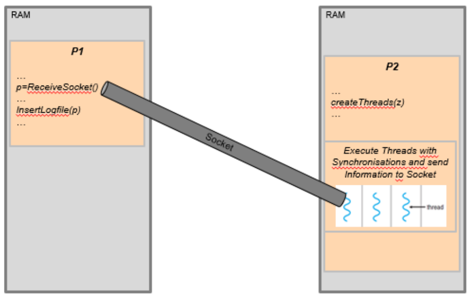
We don't have to worry about matching the threads up explicitly; that is, the threads do not necessarily know which other threads they are paired up with. The key is just that threads pass the barrier in complete sets; thus, if we examine the sequence of threads that invoke `bond` and divide them into groups of three, each group should contain one oxygen and two hydrogen threads.

Puzzle: Write synchronization code for oxygen and hydrogen molecules that enforces these constraints.

- ## Aufgaben:
  - **Synchronisation**
    Analysieren Sie das obige Problem und schreiben sie ein entsprechendes C-Programm, welches die Aufgabestellung realisiert und löst. Wählen sie passende Programmkonstrukte um parallele Strukturen, Ein- und Ausgaben und Synchronisationsmechanismen zu realisieren.

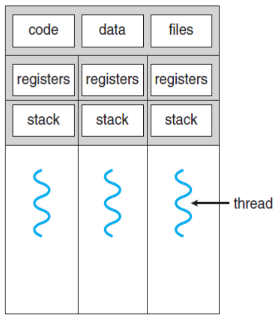
  - **Sockets**
    Erweitern Sie die obige Aufgabenstellung mit einer Socketanwendung. Mit dieser Erweiterung können z.B. Eingaben übergeben werden oder Ausgaben an einen weiteren Serverprozess übertragen werden.



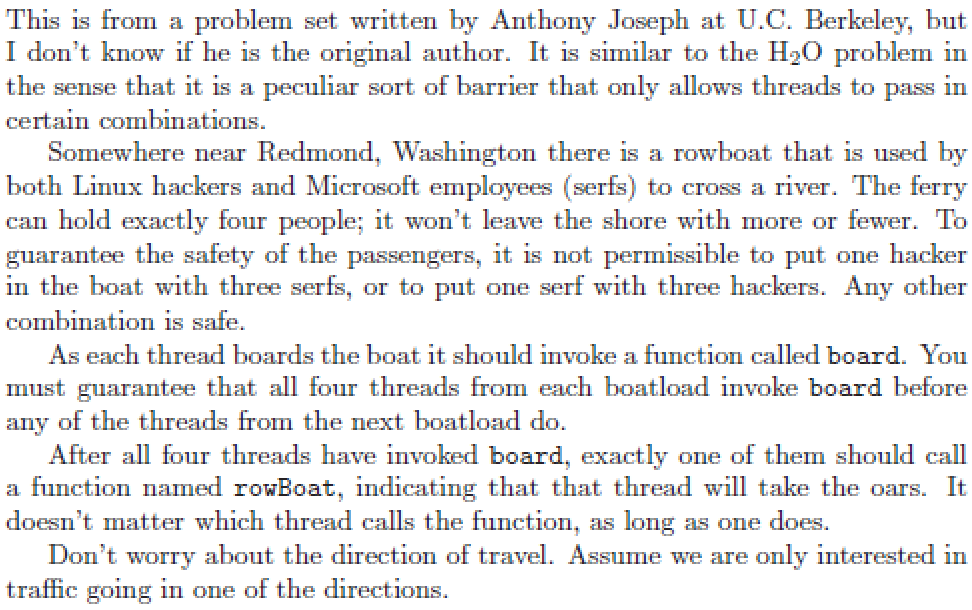


- ## Abgabe:
  Abzugeben sind ein dokumentierter SourceCode und ein Funktionsnachweis in Form von ScreenShots, Unix-Befehlen, Programmausgaben, …
  Abzugeben ist auch eine prinzipielle Beschreibung des Problems.

# Systemprogrammierung Beispiel 9:



- **River crossing problem**



This is from a problem set written by Anthony Joseph at U.C. Berkeley, but I don't know if he is the original author. It is similar to the $H_2O$ problem in the sense that it is a peculiar sort of barrier that only allows threads to pass in certain combinations.

Somewhere near Redmond, Washington there is a rowboat that is used by both Linux hackers and Microsoft employees (serfs) to cross a river. The ferry can hold exactly four people; it won't leave the shore with more or fewer. To guarantee the safety of the passengers, it is not permissible to put one hacker in the boat with three serfs, or to put one serf with three hackers. Any other combination is safe.

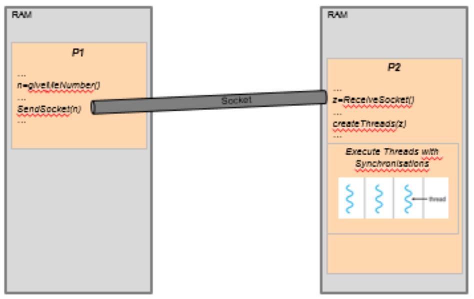
As each thread boards the boat it should invoke a function called `board`. You must guarantee that all four threads from each boatload invoke `board` before any of the threads from the next boatload do.

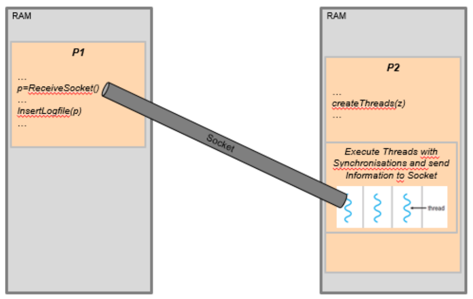
After all four threads have invoked `board`, exactly one of them should call a function named `rowBoat`, indicating that that thread will take the oars. It doesn't matter which thread calls the function, as long as one does.

Don't worry about the direction of travel. Assume we are only interested in traffic going in one of the directions.

- ## Aufgaben:
  - **Synchronisation**
    Analysieren Sie das obige Problem und schreiben sie ein entsprechendes C-Programm, welches die Aufgabestellung realisiert und löst. Wählen sie passende Programmkonstrukte um parallele Strukturen, Ein- und Ausgaben und Synchronisationsmechanismen zu realisieren.

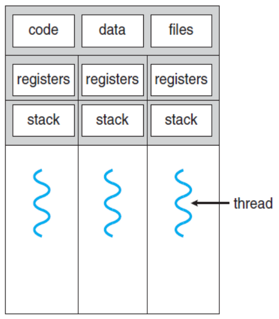
  - **Sockets**
    Erweitern Sie die obige Aufgabenstellung mit einer Socketanwendung. Mit dieser Erweiterung können z.B. Eingaben übergeben werden oder Ausgaben an einen weiteren Serverprozess übertragen werden.



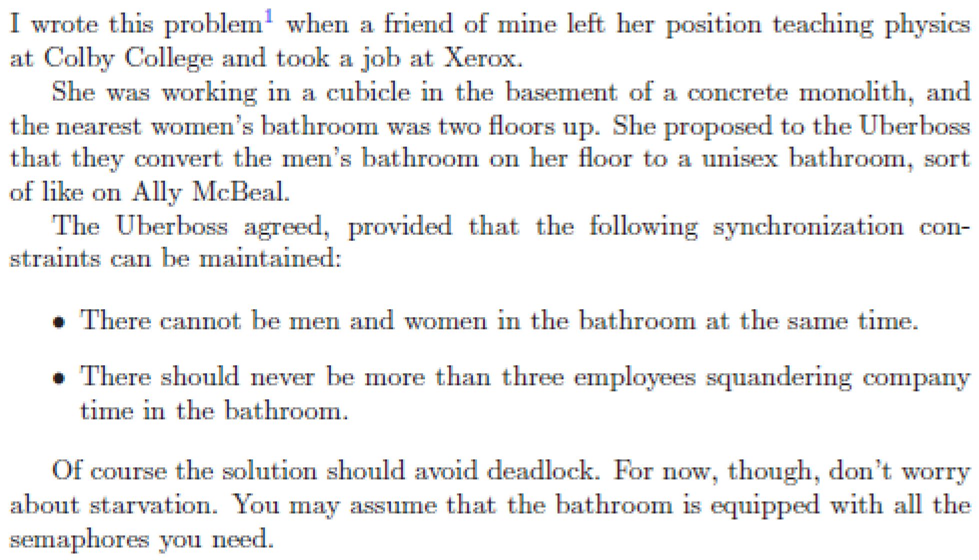


- ## Abgabe:
  Abzugeben sind ein dokumentierter SourceCode und ein Funktionsnachweis in Form von ScreenShots, Unix-Befehlen, Programmausgaben, …
  Abzugeben ist auch eine prinzipielle Beschreibung des Problems.

# Systemprogrammierung Beispiel 10:



- **The unisex bathroom problem**



I wrote this problem[1] when a friend of mine left her position teaching physics at Colby College and took a job at Xerox.

She was working in a cubicle in the basement of a concrete monolith, and the nearest women's bathroom was two floors up. She proposed to the Uberboss that they convert the men's bathroom on her floor to a unisex bathroom, sort of like on Ally McBeal.

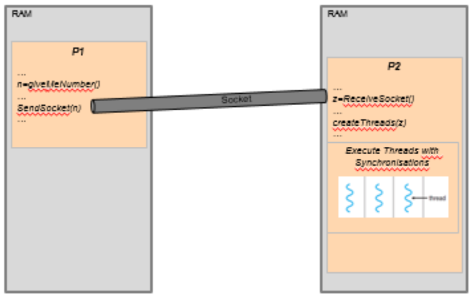
The Uberboss agreed, provided that the following synchronization constraints can be maintained:

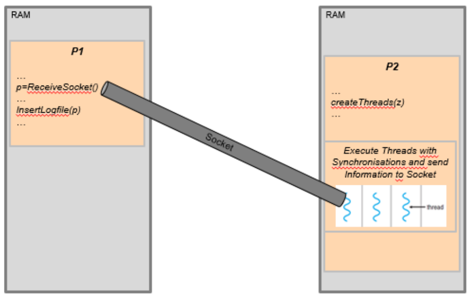
- There cannot be men and women in the bathroom at the same time.
- There should never be more than three employees squandering company time in the bathroom.

Of course the solution should avoid deadlock. For now, though, don't worry about starvation. You may assume that the bathroom is equipped with all the semaphores you need.

- ## Aufgaben:
    - **Synchronisation**
      Analysieren Sie das obige Problem und schreiben sie ein entsprechendes C-Programm, welches die Aufgabestellung realisiert und löst. Wählen sie passende Programmkonstrukte um parallele Strukturen, Ein- und Ausgaben und Synchronisationsmechanismen zu realisieren.

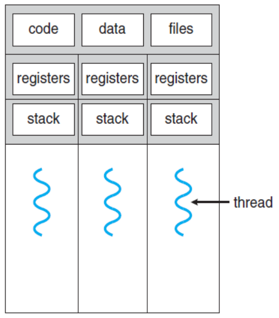
    - **Sockets**
      Erweitern Sie die obige Aufgabenstellung mit einer Socketanwendung. Mit dieser Erweiterung können z.B. Eingaben übergeben werden oder Ausgaben an einen weiteren Serverprozess übertragen werden.





- ## Abgabe:
  Abzugeben sind ein dokumentierter SourceCode und ein Funktionsnachweis in Form von ScreenShots, Unix-Befehlen, Programmausgaben, …
  Abzugeben ist auch eine prinzipielle Beschreibung des Problems.

# Systemprogrammierung Beispiel 11:



- **Baboon crossing problem**

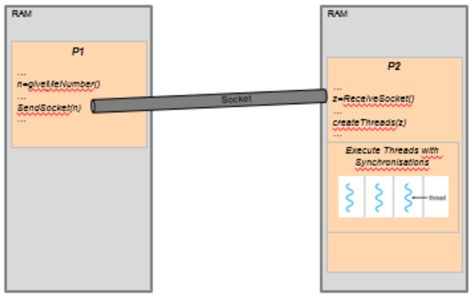
This problem is adapted from Tanenbaum's *Operating Systems: Design and Implementation* [12]. There is a deep canyon somewhere in Kruger National Park, South Africa, and a single rope that spans the canyon. Baboons can cross the canyon by swinging hand-over-hand on the rope, but if two baboons going in opposite directions meet in the middle, they will fight and drop to their deaths. Furthermore, the rope is only strong enough to hold 5 baboons. If there are more baboons on the rope at the same time, it will break.

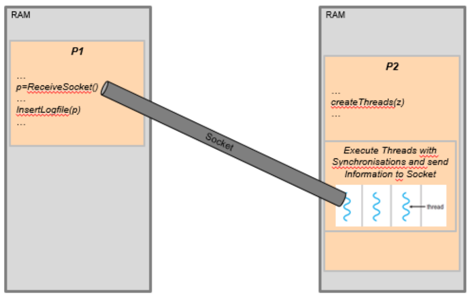
Assuming that we can teach the baboons to use semaphores, we would like to design a synchronization scheme with the following properties:

- Once a baboon has begun to cross, it is guaranteed to get to the other side without running into a baboon going the other way.
- There are never more than 5 baboons on the rope.
- A continuing stream of baboons crossing in one direction should not bar baboons going the other way indefinitely (no starvation).

- ## Aufgaben:
  - **Synchronisation**
    Analysieren Sie das obige Problem und schreiben sie ein entsprechendes C-Programm, welches die Aufgabestellung realisiert und löst. Wählen sie passende Programmkonstrukte um parallele Strukturen, Ein- und Ausgaben und Synchronisationsmechanismen zu realisieren.

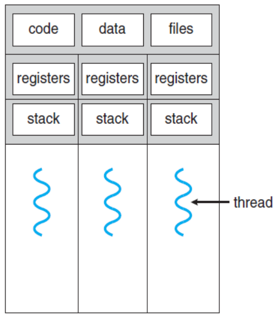
  - **Sockets**
    Erweitern Sie die obige Aufgabenstellung mit einer Socketanwendung. Mit dieser Erweiterung können z.B. Eingaben übergeben werden oder Ausgaben an einen weiteren Serverprozess übertragen werden.





- ## Abgabe:
  Abzugeben sind ein dokumentierter SourceCode und ein Funktionsnachweis in Form von ScreenShots, Unix-Befehlen, Programmausgaben, …
  Abzugeben ist auch eine prinzipielle Beschreibung des Problems.

# Systemprogrammierung Beispiel 12:



- ## The Modus Hall problem

This problem was written by Nathan Karst, one of the Olin students living in Modus Hall[2] during the winter of 2005.

> After a particularly heavy snowfall this winter, the denizens of Modus Hall created a trench-like path between their cardboard shantytown and the rest of campus. Every day some of the residents walk to and from class, food and civilization via the path; we will ignore the indolent students who chose daily to drive to Tier 3. We will also ignore the direction in which pedestrians are traveling. For some unknown reason, students living in West Hall would occasionally find it necessary to venture to the Mods.
>
> Unfortunately, the path is not wide enough to allow two people to walk side-by-side. If two Mods persons meet at some point on the path, one will gladly step aside into the neck high drift to accommodate the other. A similar situation will occur if two ResHall inhabitants cross paths. If a Mods heathen and a ResHall prude meet, however, a violent skirmish will ensue with the victors determined solely by strength of numbers; that is, the faction with the larger population will force the other to wait.

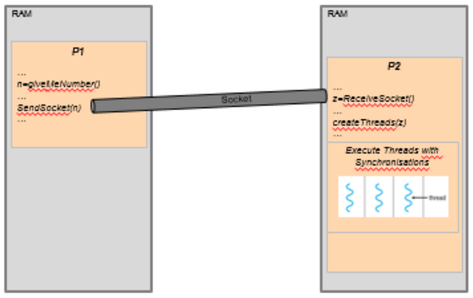
This is similar to the Baboon Crossing problem (in more ways than one), with the added twist that control of the critical section is determined by majority rule. This has the potential to be an efficient and starvation-free solution to the categorical exclusion problem.

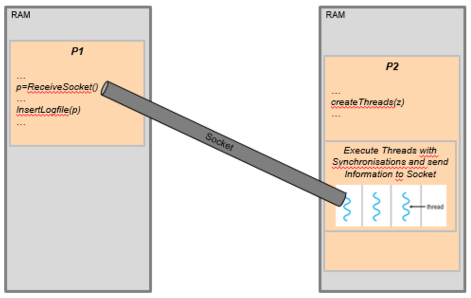
Starvation is avoided because while one faction controls the critical section, members of the other faction accumulate in queue until they achieve a majority. Then they can bar new opponents from entering while they wait for the critical section to clear. I expect this solution to be efficient because it will tend to move threads through in batches, allowing maximum concurrency in the critical section.

Puzzle: write code that implements categorical exclusion with majority rule.

- ## Aufgaben:
  - **Synchronisation**
    Analysieren Sie das obige Problem und schreiben sie ein entsprechendes C-Programm, welches die Aufgabestellung realisiert und löst. Wählen sie passende Programmkonstrukte um parallele Strukturen, Ein- und Ausgaben und Synchronisationsmechanismen zu realisieren.

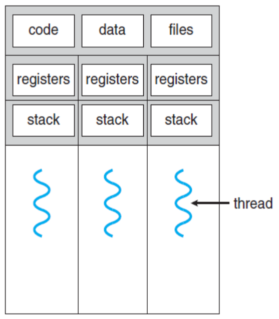
  - **Sockets**
    Erweitern Sie die obige Aufgabenstellung mit einer Socketanwendung. Mit dieser Erweiterung können z.B. Eingaben übergeben werden oder Ausgaben an einen weiteren Serverprozess übertragen werden.





- ## Abgabe:
  Abzugeben sind ein dokumentierter SourceCode und ein Funktionsnachweis in Form von ScreenShots, Unix-Befehlen, Programmausgaben, …
  Abzugeben ist auch eine prinzipielle Beschreibung des Problems.

# Systemprogrammierung Beispiel 13:



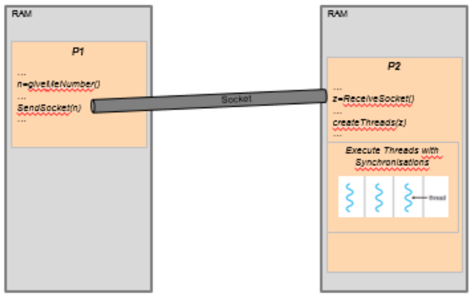
- **The sushi bar problem**

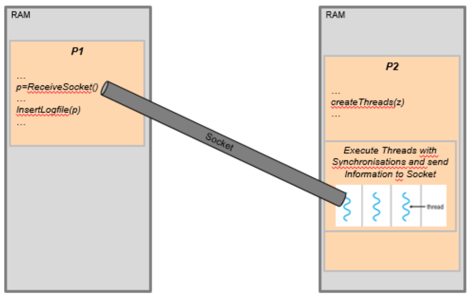
This problem was inspired by a problem proposed by Kenneth Reek [9]. Imagine a sushi bar with 5 seats. If you arrive while there is an empty seat, you can take a seat immediately. But if you arrive when all 5 seats are full, that means that all of them are dining together, and you will have to wait for the entire party to leave before you sit down.

Puzzle: write code for customers entering and leaving the sushi bar that enforces these requirements.

- **Aufgaben:**
    - **Synchronisation**
      Analysieren Sie das obige Problem und schreiben sie ein entsprechendes C-Programm, welches die Aufgabestellung realisiert und löst. Wählen sie passende Programmkonstrukte um parallele Strukturen, Ein- und Ausgaben und Synchronisationsmechanismen zu realisieren.

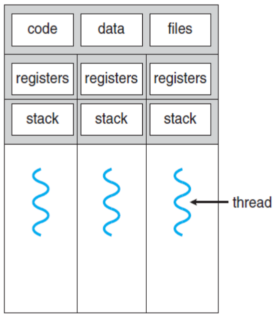
    - **Sockets**
      Erweitern Sie die obige Aufgabenstellung mit einer Socketanwendung. Mit dieser Erweiterung können z.B. Eingaben übergeben werden oder Ausgaben an einen weiteren Serverprozess übertragen werden.





- **Abgabe:**
  Abzugeben sind ein dokumentierter SourceCode und ein Funktionsnachweis in Form von ScreenShots, Unix-Befehlen, Programmausgaben, …
  Abzugeben ist auch eine prinzipielle Beschreibung des Problems.

## Systemprogrammierung Beispiel 14:



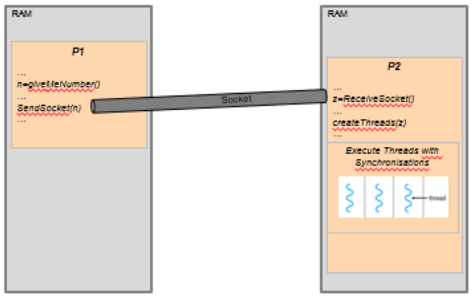
- **The child care problem**

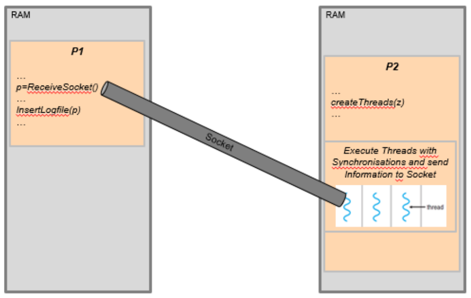
Max Hailperin wrote this problem for his textbook *Operating Systems and Middleware* [5]. At a child care center, state regulations require that there is always one adult present for every three children.

Puzzle: Write code for child threads and adult threads that enforces this constraint in a critical section.

- ## Aufgaben:
  - **Synchronisation**
    Analysieren Sie das obige Problem und schreiben sie ein entsprechendes C-Programm, welches die Aufgabestellung realisiert und löst. Wählen sie passende Programmkonstrukte um parallele Strukturen, Ein- und Ausgaben und Synchronisationsmechanismen zu realisieren.

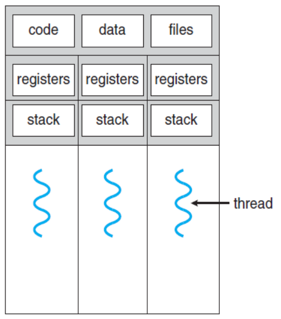
  - **Sockets**
    Erweitern Sie die obige Aufgabenstellung mit einer Socketanwendung. Mit dieser Erweiterung können z.B. Eingaben übergeben werden oder Ausgaben an einen weiteren Serverprozess übertragen werden.



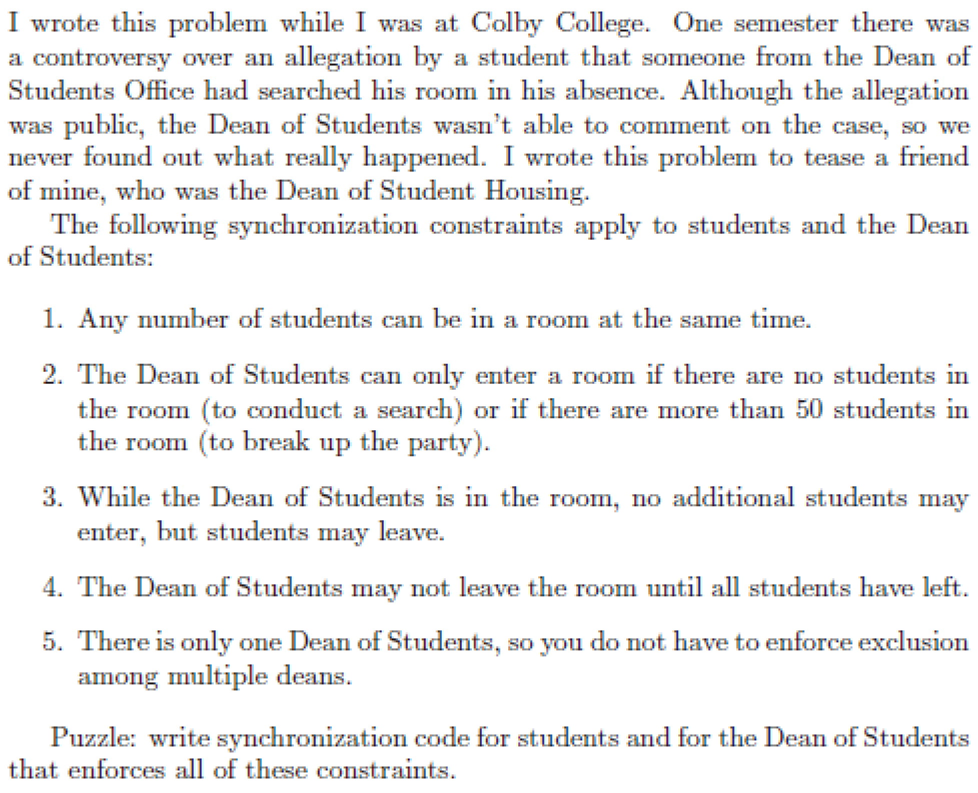


- ## Abgabe:
  Abzugeben sind ein dokumentierter SourceCode und ein Funktionsnachweis in Form von ScreenShots, Unix-Befehlen, Programmausgaben, …
  Abzugeben ist auch eine prinzipielle Beschreibung des Problems.

## Systemprogrammierung Beispiel 15:



- **The room party problem**



I wrote this problem while I was at Colby College. One semester there was a controversy over an allegation by a student that someone from the Dean of Students Office had searched his room in his absence. Although the allegation was public, the Dean of Students wasn't able to comment on the case, so we never found out what really happened. I wrote this problem to tease a friend of mine, who was the Dean of Student Housing.

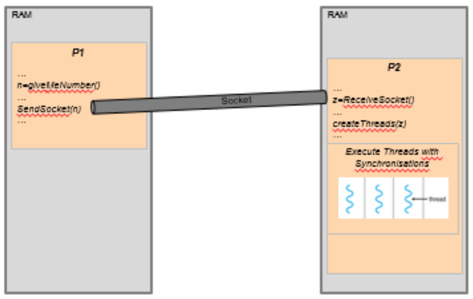
The following synchronization constraints apply to students and the Dean of Students:

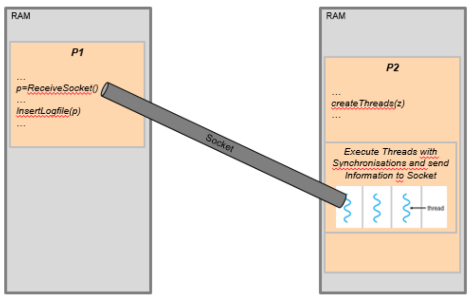
1. Any number of students can be in a room at the same time.
2. The Dean of Students can only enter a room if there are no students in the room (to conduct a search) or if there are more than 50 students in the room (to break up the party).
3. While the Dean of Students is in the room, no additional students may enter, but students may leave.
4. The Dean of Students may not leave the room until all students have left.
5. There is only one Dean of Students, so you do not have to enforce exclusion among multiple deans.

Puzzle: write synchronization code for students and for the Dean of Students that enforces all of these constraints.

- ## Aufgaben:
  - **Synchronisation**
    Analysieren Sie das obige Problem und schreiben sie ein entsprechendes C-Programm, welches die Aufgabestellung realisiert und löst. Wählen sie passende Programmkonstrukte um parallele Strukturen, Ein- und Ausgaben und Synchronisationsmechanismen zu realisieren.

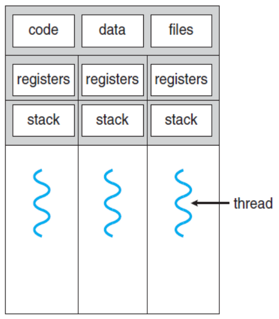
  - **Sockets**
    Erweitern Sie die obige Aufgabenstellung mit einer Socketanwendung. Mit dieser Erweiterung können z.B. Eingaben übergeben werden oder Ausgaben an einen weiteren Serverprozess übertragen werden.





- ## Abgabe:
  Abzugeben sind ein dokumentierter SourceCode und ein Funktionsnachweis in Form von ScreenShots, Unix-Befehlen, Programmausgaben, …
  Abzugeben ist auch eine prinzipielle Beschreibung des Problems.

## Systemprogrammierung Beispiel 16:



- **Dining Hall problem**

This problem was written by Jon Pollack during my Synchronization class at Olin College.

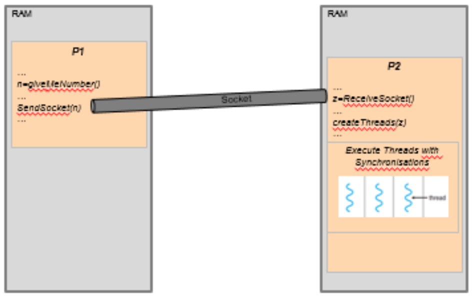
Students in the dining hall invoke `dine` and then `leave`. After invoking `dine` and before invoking `leave` a student is considered "ready to leave".

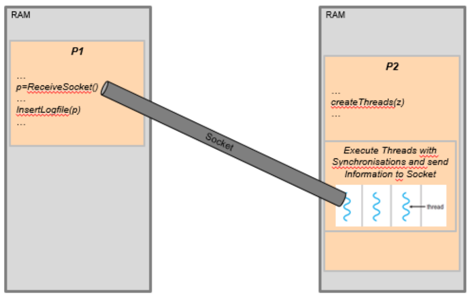
The synchronization constraint that applies to students is that, in order to maintain the illusion of social suave, a student may never sit at a table alone. A student is considered to be sitting alone if everyone else who has invoked `dine` invokes `leave` before she has finished `dine`.

Puzzle: write code that enforces this constraint.

- **Aufgaben:**
  - **Synchronisation**
    Analysieren Sie das obige Problem und schreiben sie ein entsprechendes C-Programm, welches die Aufgabestellung realisiert und löst. Wählen sie passende Programmkonstrukte um parallele Strukturen, Ein- und Ausgaben und Synchronisationsmechanismen zu realisieren.

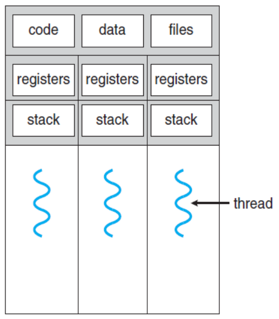
  - **Sockets**
    Erweitern Sie die obige Aufgabenstellung mit einer Socketanwendung. Mit dieser Erweiterung können z.B. Eingaben übergeben werden oder Ausgaben an einen weiteren Serverprozess übertragen werden.





- **Abgabe:**
  Abzugeben sind ein dokumentierter SourceCode und ein Funktionsnachweis in Form von ScreenShots, Unix-Befehlen, Programmausgaben, …
  Abzugeben ist auch eine prinzipielle Beschreibung des Problems.

## Systemprogrammierung Beispiel 17:



- ### The dining savages problem

A tribe of savages eats communal dinners from a large pot that can hold M servings of stewed missionary[1]. When a savage wants to eat, he helps himself from the pot, unless it is empty. If the pot is empty, the savage wakes up the cook and then waits until the cook has refilled the pot.

Any number of savage threads run the following code:

Unsynchronized savage code

```
while True:
    getServingFromPot()
    eat()
```

And one cook thread runs this code:

Unsynchronized cook code

```
while True:
    putServingsInPot(M)
```

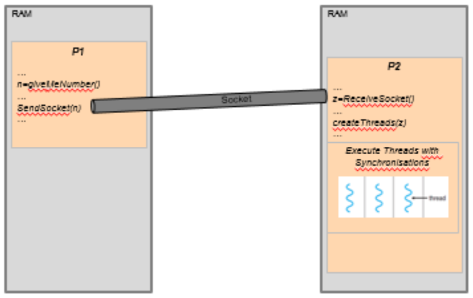
The synchronization constraints are:

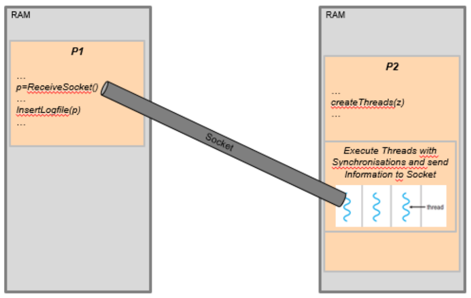
- Savages cannot invoke `getServingFromPot` if the pot is empty.
- The cook can invoke `putServingsInPot` only if the pot is empty.

Puzzle: Add code for the savages and the cook that satisfies the synchronization constraints.

- ## Aufgaben:
    - **Synchronisation**
      Analysieren Sie das obige Problem und schreiben sie ein entsprechendes C-Programm, welches die Aufgabestellung realisiert und löst. Wählen sie passende Programmkonstrukte um parallele Strukturen, Ein- und Ausgaben und Synchronisationsmechanismen zu realisieren.

    - **Sockets**
      Erweitern Sie die obige Aufgabenstellung mit einer Socketanwendung. Mit dieser Erweiterung können z.B. Eingaben übergeben werden oder Ausgaben an einen weiteren Serverprozess übertragen werden.





- ## Abgabe:
  Abzugeben sind ein dokumentierter SourceCode und ein Funktionsnachweis in Form von ScreenShots, Unix-Befehlen, Programmausgaben, …
  Abzugeben ist auch eine prinzipielle Beschreibung des Problems.