Assignments

Assignments

Assignment - In progress

Complete the form, then choose the appropriate button at the bottom.

Title
Assignment 05 - Semaphores
Due
Nov 11, 2018 11:55 pm
Number of resubmissions allowed
Unlimited
Accept Resubmission Until
Nov 11, 2018 11:55 pm
Status
In progress
Modified by instructor
Nov 1, 2018 5:03 pm

Instructions

Assignment 05 - Semaphores

Question 1 - Multi-currency conversion server

We want to allow multiple users to submit currency conversion requests to a daemon program via the terminal. You will create a client/server application for this purpose. The server program will loop in wait for incoming requests from the clients and process them.

A client program is launched with the following command:

```
$ conversion_client <server_id> <client_id> <currency> <amount>
```

Where

- `server_id` represents the identifier of the server
- `client_id` represents the identifier of the client
- `currency` represents the input currency
- and `amount` represents the amount to be converted in the target currencies.

At startup, each client creates a segment of shared memory named `/<client_id>_shm:0`
For example, a client `foo` will create a segment called `/foo_shm:0`
It then submits the conversion request to the server and displays the result upon reception

The `server` program is launched with the following command:

```
$ conversion_server <server_id>
```

where `server_id` is the identifier of the server.
At startup, the server creates a segment of shared memory named `/<server_id>_shm:0`
The program also creates N child processes: each child handles conversions for a different currency. It runs an infinite loop where it waits for its parent to initiate a new conversion, then performs this conversion and returns the result directly to the client.

NB: The protocol described above voluntarily skips out details concerning the client/server communication and synchronization. You are in charge of working out these details, describing them in your README file, and using both shared memory and semaphores as you see fit so that your application runs correctly.

Question 2 - Dining Philosophers

The dining philosophers is a classic synchronization problem. N philosophers are sitting around a large round table; there is a bowl of rice in front of each of them, and there are exactly N chopsticks arranged around the table so that each philosopher shares the chopstick that is on the right (respectively on the left) with the right (resp. left) neighbor.

Each philosopher P has the same repetitive behavior: P thinks for a while, then becomes hungry and takes the chopsticks to the left and to the right in order to eat. P stops eating after a while and puts the chopsticks back on the table to think before getting hungry again. This goes on indefinitely. In order to eat, a philosopher must have both chopsticks in hand at the same time.

Disregard the obvious health hazard presented by this behaviour, and focus on synchronization. A problem can occur when all philosophers are hungry simultaneously, and each of them picks up the chopstick to their left before picking up the chopstick to the right.
What could happen then? Explain your answer.

Program a simulation of this dinner. Each philosopher is represented by a separate process that has an identifier i (0 <i <N). All N philosopher processes share chopsticks in an array of N integers. Initially, for every i, chopsticks[i] = 1.
In order to eat, a philosopher process i must first obtain chopsticks i and (i + 1)% N. That is to say, it must verify that chopsticks [i] == chopsticks [(i + 1)% N] == 1, then set these values to 0 (chopsticks [i] = chopsticks [(i + 1)% N] = 0)

Your program must enforce safe usage of the chopsticks: in other words, two philosophers sat next to each other can not be eating at the same time.
You must also allow maximum concurrency: if an eating philosopher does prevent its two immediate neighbors to eat along, any other philosopher should be able to eat if the chopsticks to its left and right are available.