Banker's Algorithm

Aaron Alphonsus
Version 1.0
DATE: 3 April 2017

File Index

File List

Here is a list of all files with brief descriptions:

File Documentation

bank.h File Reference

Bank header file. Contains data structures to keep track of the resources.

Macros

- #define **NUMBER_OF_CUSTOMERS**   5
  Number of customers.
- #define **NUMBER_OF_RESOURCES**   3
  Number of resources.
- #define **MAX_SLEEP_TIME**   5
  Maximum time (in seconds) to sleep.

Variables

- pthread_mutex_t **mutex_lock**
  Mutex lock.
- int **available** [**NUMBER_OF_RESOURCES**]
  The available amount of each resource.
- int **maximum** [**NUMBER_OF_CUSTOMERS**][**NUMBER_OF_RESOURCES**]
  The maximum demand of each customer.
- int **allocation** [**NUMBER_OF_CUSTOMERS**][**NUMBER_OF_RESOURCES**]
  The amount currently allocated to each customer.
- int **need** [**NUMBER_OF_CUSTOMERS**][**NUMBER_OF_RESOURCES**]
  The remaining need of each customer.
- int **customer_id** [**NUMBER_OF_CUSTOMERS**]
  Numeric id of each customer.

Detailed Description

Bank header file. Contains data structures to keep track of the resources.

Author:
Aaron Alphonsus

Date:
3 April 2017

Macro Definition Documentation

#define MAX_SLEEP_TIME   5

Maximum time (in seconds) to sleep.

#define NUMBER_OF_CUSTOMERS   5

Number of customers.

#define NUMBER_OF_RESOURCES   3

Number of resources.

Variable Documentation

int allocation[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES]

The amount currently allocated to each customer.

int available[NUMBER_OF_RESOURCES]

The available amount of each resource.

int customer_id[NUMBER_OF_CUSTOMERS]

Numeric id of each customer.

int maximum[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES]

The maximum demand of each customer.

pthread_mutex_t mutex_lock

Mutex lock.

int need[NUMBER_OF_CUSTOMERS][NUMBER_OF_RESOURCES]

The remaining need of each customer.

bankers.c File Reference

Main file for the Banker's Algorithm.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <errno.h>
#include "bank.h"
#include "customer.h"
```

Functions

- void **init** (char *argv[])
- void **create_customers** ()
- int **main** (int argc, char *argv[])

Variables

- pthread_t **customers** [**NUMBER_OF_CUSTOMERS**]
  Threads representing customers.

Detailed Description

Main file for the Banker's Algorithm.

The program begins by initializing the required synchronization objects and data structures. Then, it creates a number of threads, each representing a customer. These threads run the **customer_loop()** which contains the main logic behind the Banker's Algorithm simulation.

Compilation Instructions: make

Run: ./bankers [num_resource1, num_resource2, ..., num_resourceN]

Author:

Aaron Alphonsus

Date:

3 April 2017

Function Documentation

void create_customers ()

Creates a thread for each customer to execute **customer_loop()**. Passes in the array of customer ids to give an identity to each customer thread.

Loop through defined number of customers, creating customer threads

void init (char * *argv*[])

Initializes matrices that keep track of the resource utilization. Also initializes the mutex.

Parameters:

in	*argv*	Vector of the command-line arguments

Initialize mutex

Initialize available array with values passed in via command-line

Initialize allocation matrix (0 initially)

Initialize maximum matrix using available array as a bound

Initialize need matrix. (need = maximum) initially since allocation = 0

Initialize customer id array

int main (int *argc*, char * *argv*[])

Makes function call to initialize data structures, and a call to the display function to print this initial state to the console. It then makes a function call to create customer threads and execute the threaded function **customer_loop()**, followed by the joining of these threads.

Parameters:

in	*argc*	Integer count of the command-line arguments
in	*argv*	Vector of the command-line arguments

Returns:

0 Indicates normal termination of main.

Seed random generator

Call function to initialize resource arrays and the mutex lock

Call function to display initial state

Call function to create customer threads to execute **customer_loop()**

Join customer threads

Variable Documentation

pthread_t customers[NUMBER_OF_CUSTOMERS]

Threads representing customers.

customer.c File Reference

Defines the behavior of the customer threads.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include "bank.h"
```

Functions

- void **display** ()
  Displays state of the system.
- int **safety_test** ()
  Checks for safe state.
- int **request_resources** (int customer_num, int request[])
  Check whether request for resources can be granted.
- int **release_resources** (int customer_num, int release[])
  Returns resources to available pool.
- void * **customer_loop** (void *param)
  Customer function loops continually, requesting and releasing resources.

Detailed Description

Defines the behavior of the customer threads.

The customer threads spin in a continuous loop, requesting and releasing random numbers of resources. The banker evaluates the request and makes sure that the resulting system is not unsafe. A display function helps keep track of the changes that happen at each step.

Author:

Aaron Alphonsus

Date:

3 April 2017

Function Documentation

void* customer_loop (void * *param*)

Customer function loops continually, requesting and releasing resources.

This function simulates the behavior of each customer. The customers request some amount of resources, wait and release some amount of resources. This happens in a continuous loop

Parameters:

in	*param*	Pointer to an int pointer to the student id

Declare local helper variables

Seed random generator

Loop continuously

Create random request array (% need[customer_id][])

Request resources

Sleep for random amount of time

Create random release array (% alloc[customer_id][])

Release resources

Sleep for random amount of time

Reset local variables

void display ()

Displays state of the system.

Function to print the state of the system out to the console in a tabular format.

Declare array for resource type

Print header

Print resource type header

Print data structures keeping track of resources

int release_resources (int *customer_num*, int *release*[])

Returns resources to available pool.

This function releases a random number of resources bounded by values in the allocation array. Since this function modifies shared resources, we make sure that we acquire the mutex lock before making changes, and release it once we are done.

Parameters:

in	*customer_num*	Holds the customer id
in	*release*	The random release array generated

Returns:

0 if successful

Acquire mutex lock

Print release array

Update resource arrays

Display current state

Release mutex lock before returning

int request_resources (int *customer_num*, int *request*[])

Check whether request for resources can be granted.

This function requests for a random number of resources bounded by values in the need array. Since this function modifies shared resources, we make sure that we acquire the mutex lock before making changes, and release it once we are done. The request for resources can fail if there aren't enough resources available or if granting the resources causes the system to be placed into an unsafe state. Appropriate messages are provided in each case.

Parameters:

in	*customer_num*	Holds the customer id
in	*request*	The random request array generated

Returns:

0 if granted, -1 if not

Acquire mutex lock

Print request statement and array

Check if request is less than available resources

Display message if resources unavailable and return -1

Make changes to available, allocation and need

Call safety test

If request granted, print message.

If unsafe, rollback changes

Release mutex lock before returning

int safety_test ()

Checks for safe state.

Contains the algorithm that determines whether the system is in a safe state or not. (Reference: 7.5.3.1 - Operating System Concepts - Silberschatz)

Returns:

0 if safe, -1 if not

Declare work and finish arrays (along with helper variables)

Initialize work and finish arrays

For each 'false' finish element, check if need <= work

Once the algorithm is finished, look for false values in the finish array which indicates whether the state is unsafe

customer.h File Reference

Customer header file. Contains function prototypes to request and release resources (along with helper functions)

Functions

- void * **customer_loop** (void *param)
  Customer function loops continually, requesting and releasing resources.
- int **request_resources** (int customer_num, int request[])
  Check whether request for resources can be granted.
- int **release_resources** (int customer_num, int release[])
  Returns resources to available pool.
- int **safety_test** ()
  Checks for safe state.
- void **display** ()
  Displays state of the system.

Detailed Description

Customer header file. Contains function prototypes to request and release resources (along with helper functions)

Author:

Aaron Alphonsus

Date:

3 April 2017

Function Documentation

void* customer_loop (void * *param*)

Customer function loops continually, requesting and releasing resources.

This function simulates the behavior of each customer. The customers request some amount of resources, wait and release some amount of resources. This happens in a continuous loop

Parameters:

in	*param*	Pointer to an int pointer to the student id

Declare local helper variables

Seed random generator

Loop continuously

Create random request array (% need[customer_id][])

Request resources

Sleep for random amount of time

Create random release array (% alloc[customer_id][])

Release resources

Sleep for random amount of time

Reset local variables

void display ()

Displays state of the system.

Function to print the state of the system out to the console in a tabular format.

Declare array for resource type

Print header

Print resource type header

Print data structures keeping track of resources

int release_resources (int *customer_num*, int *release*[])

Returns resources to available pool.

This function releases a random number of resources bounded by values in the allocation array. Since this function modifies shared resources, we make sure that we acquire the mutex lock before making changes, and release it once we are done.

Parameters:

in	*customer_num*	Holds the customer id
in	*release*	The random release array generated

Returns:

0 if successful

Acquire mutex lock

Print release array

Update resource arrays

Display current state

Release mutex lock before returning

int request_resources (int *customer_num*, int *request*[])

Check whether request for resources can be granted.

This function requests for a random number of resources bounded by values in the need array. Since this function modifies shared resources, we make sure that we acquire the mutex lock before making changes, and release it once we are done. The request for resources can fail if there aren't enough resources available or if granting the resources causes the system to be placed into an unsafe state. Appropriate messages are provided in each case.

Parameters:

in	*customer_num*	Holds the customer id
in	*request*	The random request array generated

Returns:

0 if granted, -1 if not

Acquire mutex lock

Print request statement and array

Check if request is less than available resources

Display message if resources unavailable and return -1

Make changes to available, allocation and need

Call safety test

If request granted, print message.

If unsafe, rollback changes

Release mutex lock before returning

int safety_test ()

Checks for safe state.

Contains the algorithm that determines whether the system is in a safe state or not. (Reference: 7.5.3.1 - Operating System Concepts - Silberschatz)

Returns:

0 if safe, -1 if not

Declare work and finish arrays (along with helper variables)

Initialize work and finish arrays

For each 'false' finish element, check if need <= work

Once the algorithm is finished, look for false values in the finish array which indicates whether the state is unsafe