Betriebssysteme Assignment 1

Design Document 1

Group 13

Werner Schuster, Andreas Rath, Kerstin Pötsch, Wolfgang Pototschnik

Version 1.1, 8.11.2002

Contents

1 Overview

The work for this assignment is split into changes to the thread system itself, consisting of the implementation of

- Locks (see section 2.1), (Task 1)
- Condition Variables (see section 2.2), (Task 1)
- Join functionality for threads (see section 2.3), (Task 3)

and the implementation of tests of the system, such as

- implementation of the Sleeping Barber Problem (see section 3.1), (Task 2)
- test code for the Join functionality, (part of Task 2)
- assorted tests to assure correct implementation of the changes.

1.1 Changes from Version 1.0

Updated sections include are:

- 2.2.2 (new checks)
- 2.3.2 (new methods)
- 2.3.3 (changes)
- 3.1 (new commandline options,...)
- 3.2 (new test)
- 3.3.1 (ConditionVarTest)

2 Changes to the thread system

A thorough definition of the changes to the thread system.

2.1 Locks

The class `Lock` is defined in the file `synch.h`, its methods are implemented in `synch.cc`.

2.1.1 Fields

- Thread * currentHolder = NULL
- int recursionCount = 0
- bool isLocked = false
- List * waitingThreads

2.1.2 Methods

- `Acquire()` acquires a lock;
  - turn off interrupts (and store old interrupt state)
  - if not locked then lock and store reference to holding thread
    - if the acquiring thread is the thread that holds the log increase the recursion counter but don't block
    - else put the thread onto the waiting thread queue and send the thread to sleep
  - reset interrupts to stored state
- `Release()`
  - turn off interrupts (and store old interrupt state)
  - if the lock is held by the calling thread then
    - if other threads are waiting for the lock then choose the first one and put it on the ReadyToRun list again
    - else release the lock, unless the recursion count is greater than zero, and in that case decrement the recursion counter
  - reset interrupts to stored state
- isHeldByCurrentThread()
  returns true if the Lock is held by the current thread, else it returns false

2.2 Condition Variables

The class `Lock` is defined in the file `synch.h`, its methods are implemented in `synch.cc`.

2.2.1 Fields

- List * waitingThreads = new List()
  holds the threads that are waiting for the condition
- Lock * conditionLock = NULL

2.2.2 Methods

- `Wait(Lock * conditionLock)` blocks the calling thread until some other thread calls Signal or Broadcast on this condition variable
  - turn off interrupts (and store old interrupt state)
  - checks if conditionLock is held by current thread (if not: show error message and ASSERT)
  - Append the calling thread to the waitingThreads list
  - store the conditionLock
  - Release the conditionLock
  - call Sleep on the currentThread
  - re-Acquire the conditionLock
  - reset interrupts to stored state
- `Signal(Lock * conditionLock)`
  - turn off interrupts (and store old interrupt state)
  - checks if conditionLock is held by current thread (if not: show error message and ASSERT)
  - remove the next thread from the waitingThreads list and put it on the ReadyToRun list again
  - reset interrupts to stored state
- `Broadcast(Lock * conditionLock)`
  - turn off interrupts (and store old interrupt state)
  - checks if conditionLock is held by current thread (if not: show error message and ASSERT)
  - remove all threads from the waitingThreads list and put them on the ReadyToRun list again
  - reset interrupts to stored state

2.3 Join

Implementation of the Join method required changes to the thread system in the `thread.h` and `thread.cc` files. The Join method allows a parent thread to wait for one specific child thread to finish.

2.3.1 Fields

The following fields were added to the Thread class:

- Thread * parent
- List * childrenList
  List of children forked by this thread
- bool isJoinable = false
- bool hasBeenJoined = false

2.3.2 New Methods

- `Thread::Join(Thread * pThread)`
  pThread is the thread, that the calling thread wants to wait for;
  - turn off interrupts (and store old interrupt state)
  - if pThread is in the children list of the calling thread
    - set hasBeenJoined = true (the field in the child thread)
    - put the current thread to Sleep
  - reset interrupts to stored state
- `Thread::Thread(char * name, bool joinable)`
  basically the same as the existing constructor, but it is possible to set the joinability (ie. if it is possible, to call Join on this thread object) with the joinable argument;
- `Thread::isChild(List * list, Thread * thr)`
  checks if item is containend in childrenList this needs to be done, since the List does not have any methods to do that (List::Mapcar(void*) is not easily usable with C++);
- `Thread::InsertIntoList(void * item)`
  inserts the item into childrenList using List::Append
- `Thread::RemoveFromList(void * item)`
  removes item from the childrenList
- `RemoveItem(List * list, void * item)`
  function that inserts an item into list (since the List class does not provide proper methods for this; this should probably be moved to the List class;

2.3.3 Changes in existing methods

The described actions are *added* to the original behaviour of these methods.

- `Thread(char * name)`
  initializes the children List;
- `Fork`
  adds the new thread to the children list of the parent; sets the parent=currentThread;
- `Finish()`
  if (hasBeenJoined) => arranges for the parent thread to be put on the Ready-ToRun list again

3 Tests for the thread system

All the test output contains at least the string "test: OK" or "test: FAILED". Since the tests emit all kinds of information (sometimes a lot), you can grep/look for these strings in the output for the test result.

3.1 Sleeping Barber Problem

The code for the Sleeping Barber problem is implemented in the file `threadtest.cc`. Since it is a rather known problem, it will not be further discussed here (for a proper description refer to a textbook on operating systems).
The test is started by invoking the `TestBarbershop` method.
To invoke the test on the command line, use one of these options:

- ./nachos –barber (that's 2 dashes!)
  for the normal test (30 customers)
- ./nachos –barber__stress (that's 2 dashes!) for the stress test (30000 customers); CAUTION: this can take a long time on slow computers!

3.1.1 Fields

- Condition * cvWaitingCustomers = new Condition("WaitingCustomers");
- Condition * cvBarberChair = new Condition("BarberChair");
- Lock * lockBarber = new Lock("lockBarber");
- Lock * lockWaitingCustomers = new Lock("lockWaitingCustomers");
- int bool barberChairFree = true;
- int numWaitingChairs
- int numCustomers
- int numWaitingCustomers = 0
- bool shopOpen = false,
  this is set to false if the test done and must stop, ie the Barber checks this field and returns after he has finished the last customer
- int numCustomerServedCustomers = 0
- int numRefusedCustomers = 0
- int numBarberServedCustomers = 0

3.1.2 Methods

- `TestBarbershop(int num_chairs, int num_customers )`
  numWaitingChairs = num_chairs, numCustomers = num_customers

  1. initializes everything (Variables, Barber,...)
  2. forks Barber thread
  3. forks CustomerDispatcher Thread, thereby starting the test
  4. calls Join on the Barber thread (waiting for the test to finish)
  5. performs some checks to see if the test has failed or not, ie. if numCustomerServed + numRefusedCustomers == numCustomers

- `CustomerDispatcher(int num_customers)`
  consists of a loop that runs `num_ customers` times and dispatches `num_ customers` Customer threads. The Customers are dispatched (meaning, new Customer threads are created and forked) at (if possible) random times. Thread::Yield should be called to allow the Barber thread to run.

- `Customer(void)`
  tries to get a seat (leaves if none is available) and cvWaitingCustomers− >Signal(Lock) and while (!barberChairFree) waits using cvBarberChair− >Wait() until it can get exclusive access to the lockBarberChair; while the barberChair is not free; calls GetHairCut() after gaining access to the barberChair;

- `Barber(void)`
  Runs in a loop and checks the shopOpen field (if that is false and there are no customers, the method returns). Calls cvWaitingCustomers− >Wait(Lock) while there are no waiting customers, otherwise acquires the lockWaitingCustomers, calls the CutHair() method and afterwards the cvBarberChair− >Signal(Lock) (before releasing the lockWaitingCustomers;

- `GetHaircut(void)`
  numCustomerServedCustomers++, called by Customer after acquiring lockBarberChair;

- `CutHair(void)`
  numBarberServedCustomers++, numWaitingCustomers–; called by Barber after acquiring lockWaitingCustomers;

- `LeaveShop(void)`
  numRefusedCustomers++; release lockWaitingCustomers, return from Customer method;

3.1.3 Threads

- `Customer_Dispatcher`: 1 thread
  forks new Customers

- `Barber`: 1 thread
- `Customer`: num_customer threads

3.2 Join

3.2.1 Fields

- int numJoinTests

3.2.2 Methods

- JoinTest
  Checks the Join method.; can be invoked by calling **./nachos –join** (that's 2 dashes!) It uses a simple test that launches a thread Gepetto which then launches a child thread Pinoccio and joins it. If the test succeeds, thread Gepetto gets the control again. The JoinTest method forks the parent thread.
- JoinTestParent the method for the parent thread, that forks the child thread and Joins it. Emits the test result output.
- JoinTestChild the method for the child thread.

3.3 Other Tests

Other tests that are required, eg. to make sure the Acquire/Release methods work recursively, ...

3.3.1 Methods/Tests

- `LockTest()`
  checks the Locks; can be invoked by calling **./nachos –lock** (that's 2 dashes!); it first checks the Locks with 2 threads, then invokes the recursiveLockTest specifying the desired recursion depth;
  - `recursiveLockTest(Lock * lock, int recLeft)`
    acquires the lock, decrements recLeft and if recLeft > 0 calls itself; after that it releases the lock again;
- `ConditionVarTest()`
  checks the Condition vars; can be invoked by calling **./nachos –cond** (that's 2 dashes!); creates 3 threads, where one has to Wait() for the first one to Signal(), then sends out a Broadcast() to wake up the remaining 2 threads;
  - condTestCode(int id)
    Launches 3 threads, tests Signal on one and Broadcast on the others. This is achieved with a new global var testCount which is set to 3, and

then decremented by every thread that was woken up by Signal or Broadcast; the test is OK if the testCount var is 0 at the end (checked by the ConditionVarTest);