Project — Mneme

Weight: 21% of your course grade.

Preliminaries

As with all work in this course, a small part of the assessment for the project reflects how your work is presented. Your work must be easy to read. This means:

- Your work must be clear, succinct, grammatically correct and correctly spelled.
- Your work must be laid out sensibly. In text, lines should be short enough to be readable (i.e., do not fill the entire width of the page with text—appropriate use of margins and other blank space on a page can dramatically improve the readability of your work).
- Your program code should be indented sensibly, and be easily read by anyone familiar with UNIX and C/C++. Well-written code that implements straightforward functionality may not need any additional comments.
- Unless there is a compelling reason to do otherwise, all pages should be on letter-size paper, with content in the portrait orientation. Pages should be held together with a single staple in the top lefthand corner.
- Avoid unnecessary embellishment—you should not use presentation folders.

The late policy for this project is as follows: No project components will be accepted late.

The project *must* be undertaken by groups of two, three or four people. Assessment and initial grades will be independent of group size, but I will scale your final percentage grade based on your group size, (with 0% and 100% being fixed points in the scaling). Thus, I expect larger groups to produce a more polished design with greater functionality and higher performance than projects produced by smaller groups. All groups will be expected to produce well-written source code and clear documentation.

The project may be refined or clarified in class or email.

Overview

One method for performing parallel computation is to share a common memory area between processes. But shared memory ordinarily requires that programs sharing memory run on a single machine, which reduces the amount of parallelism possible.

In this project, you will develop *Mneme*, a system that allows data to be shared easily between multiple machines. To application programmers, Mneme appears to provide shared memory, but internally Mneme 'fakes' shared memory by communicating with a central server using message passing—no memory is physically shared.

You will be developing a prototype "proof of concept" implementation of Mneme. As such, your implementation will actually run on a single machine.

The prototype will be done in three phases: development of a shared variable server and library, development of a paging server and library, and development of your own enhancements to Mneme.

Phase 1: Shared Integer Variables

Your Mneme library will provide a class, *SharedInt*, with the following operations:

SharedInt(*const char* * name) *constructor*

Creates an integer shared variable whose global name is name. All *SharedInt* objects with the same global name appear to operate on the same piece of global data.

int read() *member function*

Reads the current value stored in the shared variable.

void write(*int* newValue) *member function*

Writes a new value into the shared variable.

void lock() *member function*

Locks the shared variable. While the shared variable is locked, other processes will wait until it is unlocked before they receive access. (It is an error to lock a variable that is already locked.)

void unlock() *member function*

Unlocks the shared variable. (It is an error to unlock a variable that is not locked.)

A Trivial Implementation

Below, I outline one possible implementation strategy for the *SharedInt* class, based on the code provided to you.

The shared variable server maintains the variables in two possible states, *checked in* and *checked out*, and supports two kinds of messages: (1) a *fetch operation* (initiated by a FetchMessage) checks out a variable and sends it to a process; (2) a *store operation* (initiated by a StoreMessage) receives a new value for a variable and stores it, marking the variable as checked in.

If a request is made to check out a variable which is already checked out, the request is put into a queue for that variable. When the variable is checked back in, it will be sent out to the first outstanding requester.

The server provided to you (in message-server.cc) supports fetch and store operations, but does not keep track of whether variables are checked out or checked in. It also does not store a queue of waiting requests for the variable. Extending it to support this additional state is fairly straightforward.

Given a server with the above behaviour, the following is an outline of an implementation of *SharedInt*.

LocalPort ourPort *private static member*

A class-wide variable used to communicate with the server.

RemotePortAddress server *private static member*

The port address of the shared variable server.

string name *private member*

The name of the variable (as provided in the constructor).

bool locked *private member*

Set when the shared variable has been checked out from the server (by lock, described below) and will need to be checked back in later (by unlock, described below).

int cache *private member*

A cached local copy of the variable, used when the variable is in the locked state.

SharedInt(*string* varName) *constructor*

Initializes locked to false. Initializes name to varName.

int read() *member function*

If locked is false, send a *FetchMessage* to the server to check the variable out, and immediately send a *StoreMessage* to check it back in unchanged. Return the value received/sent.

If locked is true, return the value stored in cache.

void write(*int* newValue) *member function*

If locked is false, send a *FetchMessage* to the server to check the variable out, and immediately send a *StoreMessage* to check it back in with the new value.

If locked is true, store the new value in cache.

void lock() *member function*

If locked is false, send a *FetchMessage* to the server to check the variable out, and store its value in cache.

If locked is true, throw an exception.

void unlock() *member function*

If locked is true, send a *StoreMessage* to the server to check the variable back in, giving it the value stored in cache.

If locked is false, throw an exception.

Phase 2: Shared Memory Pages

In this phase, you will add support to the Mneme library to support the following two operations:

void * shmem_alloc(*size_t* size, *const char* * name) *function*

Allocates an area of shared memory. All allocated areas with the same name will be shared and must have the same size. The function returns the address of the allocated memory area.

void shmem_free(*void* * address, *size_t* size) *function*

Deallocates an area of shared memory. The parameter address must point to the start of an area allocated by shmem_alloc and size must match the original size of the allocation. No part of the area may be locked.

void shmem_lock(*void* * address, *size_t* size) *function*

Locks an area of shared memory. While an area of memory is locked, it is guaranteed that no other process will be able to update that area (they will block until the area is unlocked). Only one portion of the shared memory area may be locked at a time. The portion of memory locked may be larger than that requested (up to the entire shared memory area). The range from address to address + size must fall inside an area of memory allocated by shmem_alloc.

void shmem_unlock(*void* * address, *size_t* size) *function*

Unlocks an area of shared memory. While an area of memory is locked, it is guaranteed that no other process will be able to update that area (they will block until the area is unlocked). Only one portion of the shared memory area may be locked at a time. The range from address to address + size must correspond exactly to an area of memory locked by a call to shmem_lock.

A Trivial Implementation

SharedPage *private class*

Analogous to the *SharedInt* class, this class provides state for pages. The *SharedPage* class stores the state of a particular page (i.e., whether it is checked out from the server or not)—unlike the *SharedInt* class, shared data (the page contents) are not store inside the object, but elsewhere in memory. The member functions read and write in *SharedInt* have no equivalent in *SharedPage* because reading and writing is performed directly on the shared memory area.

*map<void *, SharedPage>* page_table *private data*

This data structure maps the starting addresses of pages of memory to classes to *SharedPage* objects that describe the page.

void * shmem_alloc(*size_t* size, *const char* * name) *function*

The first call to shmem_alloc installs handle_fault (described below) as a protection fault handler.

shmem_alloc allocates an area of memory (using palloc, rounding the size of the request to whole pages) to serve as the shared memory area, and sets the protection on each page to NOACCESS. It creates a *SharedPage* object for each allocated page, storing them in page_table.

void shmem_free(*void* * address, *size_t* size) *function*

shmem_free deletes each page in the address range from page_table, and then deallocates the area of memory (using pfree).

void shmem_lock(*void* * address, *size_t* size) *function*

To lock an area of shared memory, shmem_lock calls the lock member function on the *SharedPage* object corresponding to each page in the range.

void shmem_unlock(*void* * address, *size_t* size) *function*

To unlock an area of shared memory, shmem_unlock calls the unlock member function on the *SharedPage* object corresponding to each page in the range.

int handle_fault(*void* * address) *private function*

This function is called if the user code attempts to access an area of memory for which we do not have a checked-out page from the shared memory server. The code is similar to shmem_lock.

However, after the access has been made, the page needs to be unlocked so that it will be sent back to the server. This can be achieved using the setitimer system call to cause a SIGALRM signal to be sent to the process at some point in the near future. Additionally, we arrange that SIGALRM will be caught by handle_alarm.

void handle_alarm(*int* signo) *private function*

This function is called when the time limit expires on a page that was checked out by handle_fault. This handler checks the page back in with the shared memory server.

(Implementing this functionality is complicated somewhat by the fact that several pages may be checked out at different times by handle_fault, so some kind of queue will be necessary).

Phase 3: Enhancements

You are free to make any enhancements you feel appropriate. For example, you could

- Remove limitations given in the specification. (E.g, the specification says that you may only have one lock at a time. You could allow multiple locks, provided they cover different areas of virtual memory.)
- Add some statistics-gathering code to the shared memory server and an interface to allow clients to request those statistics from the server.
- Allow multiple processes to checkout data for reading if there are no processes writing.
- Define parts of the specification that are poorly defined. (E.g., the specifications above do not specify whether shared data persists if there are no processes currently using it—you could determine a sensible policy and implement it.)
- Extend the code so that it runs over a real network (don't try this unless the members of your group are familiar with network programming).

I recommend that you mull over possible extensions throughout your implementation work for phases one and two, but do not decide on your extensions until after your work on those components is complete. I also recommend that you always keep a code base that works at some level. That way, even if you have not finished your Phase 3 code by the end of the semester, you will still be able to show polished Phase 2 code.

Your Implementation

Above I have defined the functionality required for Phase 1 and Phase 2 of Mneme. You *must* implement those interfaces. During the final weeks of the project, I will provide test programs for your code to run, and those programs will assume that your Mneme library provides the above interfaces and that they function as specified.

You do not, however, have to use the suggested implementation strategy, nor do you have to use the source code I have provided. However, you will not receive credit for "reinventing the wheel" by needlessly rewriting code that I have given you. (You may still want to rewrite the code, but there should be tangible benefits from any rewrite.)

Your implementation must be written in C or C++. The provided codebase and required interfaces suggest a C++ interface, but an alternative specification for C coding is available on request. C++ implementation is recommended, however.

You implementation must run under either NextStep 3.x, Solaris 2.6 (or higher), or Linux (on Intel PCs)[1]. NextStep and Solaris 2.6[2] are the preferred platforms, and are the only platforms directly supported by the course. It is possible to write portable code which will compile

1. Linux on other platforms may require kernel changes to allow the code for protection fault-handling code to work correctly.

2. Note that the CSIL machine alrisha currently runs Solaris 2.5, which is not supported.

under all three operating systems, allowing you to develop code at home and demonstrate it using available machines. If you choose not to write portable code, you are responsible for ensuring that suitable machines are available to properly demonstrate your code at the appointed time—thus, if you use Linux, you may need to bring in a laptop computer with Linux installed.

Provided Code

I have provided you with code, available both from the Web site and the course directory, that provides a starting point for developing Mneme.

It includes a Makefile for use with the gmake compilation management tool. I recommend that you use this tool if you choose to use my code. There are also comments in the Makefile that describe how to add files to the Mneme library and how to add additional programs to be built.

All of the classes and C modules provided to you have comments in the header files that describe them. Several of the provided components have complex internals whose behaviour you *do not* need to understand. For example, ports.cc implements a message-passing class using Unix datagram sockets, but I do not expect you to understand how Unix datagram sockets work. Similarly, faulthandler.c contains complex machine-dependent code to catch memory protection violations. In each case, reading the header file should be sufficient.

From time to time, I may discover and correct bugs in the provided code. When I make such corrections, I will send email to the class mailing list. You are responsible for making the necessary changes to your code.

You may also use other code that you have not written yourself. Such code should be clearly attributed and it should always be clear what code you have written yourself. Finding freely available code and adapting it for your own use is an important skill and may be rewarded with suitable credit.

Project Stages, Deliverables and Deadlines

- **Phase 1**, demonstrated during my office hours, Wednesday, June 16, Friday, June 18, or Monday, June 21. (A sign-up sheet will be available.)

  (a) Complete an operational *SharedInt* class.

  (b) Develop and apply test code to check that your class performs correctly.

  (c) Collect notes made during your design and commented listings of your code. Be prepared to show me this documentation during your demonstration.

- **Phase 2**, demonstrated during my office hours, Friday, July 2, Monday, July 5, or Wednesday, July 7. (A sign-up sheet will be available.)

  (a) Complete an operational distributed–shared-memory system.

  (b) Develop and apply test code to check that your code performs correctly.

  (c) Collect notes made during your design and commented listings of your code. Be prepared to show me this documentation during your demonstration.

- **Phase 3** demonstrated during my office hours, Monday, July 19, Wednesday, July 21, or Friday, July 23. (A sign-up sheet will be available.) The report is due Monday, July 26.

  (a) Implement your enhancements to the Mneme system.

  (b) Develop and apply test code to check that your enhancements perform correctly.

(c) Collect notes made during your design and commented listings of your code. Be prepared to show me this documentation during your demonstration.

(d) Submit a report that describes your enhancements and comments on future directions. Include full source listings as an appendix with appropriate comments and highlighting so that your code is easy to identify and understand.