Assignment 1

Please read the assignment *carefully!* There are many suggestions contained in the specification that will make your life significantly easier. Furthermore, check back regularly, as we will put clarifications up here if there are misunderstandings or bugs in the spec.

DO NOT make assignment solutions available publicly online (for example, hosted on github.com).

Updates to the spec will be highlighted

NOTE

There is a FAQ.

Groups

This is a team assignment. You are expected to solve the assignment in groups of two or three. However, if you really can't work in a group you may also do it individually. The groups are posted here.

Introduction

Your task is to implement a user-space page-based distributed shared memory (DSM) system that enables a set of processes located on a collection of Linux hosts to share memory pages. Sequential consistency is to be realised using a simple multiple-reader/single-writer scheme implementing a write-invalidate policy[1].

Structure

The structure of the DSM system is as follows.

- Node processes run on separate hosts. All node processes run the same client program.
- Node processes cooperate by reading and writing memory pages in a shared virtual address space.
- Each node process can access the whole shared virtual address space, and each page is accessed at the same virtual address in all node processes (with different pages being accessed at different addresses).
- Client programs that are run by the node processes are linked to an SM library, which implements the functionality that allows access to the shared memory.
- The library handles page faults when the client attempts to read pages that are not locally available, or when it tries to write to pages that are not both available and write-enabled.
- The library also implements synchronisation functions and allows clients to allocate shared memory from the virtual address space.

The node processes are all started from a single node using a program called `dsm`. This program takes as a parameter the name of the client program executable and starts a given number of instances of the client program on separate nodes.

In this assignment a centralised allocator process manages communication and tracks the shared memory pages.

Assignment

The goal of the assignment is to implement the above system. Specifically, you are required to provide:

- an implementation of the SM library (that can be linked to a client program)
- an implementation of the allocator process
- an implementation of the `dsm` program (that is used to start up the node processes and then becomes the allocator process)

Note that you are not supposed to provide any client programs. To clarify this, it is useful for you to implement client programs in order to test your implementation, however, you are not meant to submit any of these programs. We have our own set of client programs that we will use to test your implementation (by linking them to your SM library and running them using your `dsm` program and your allocator).

Submissions will be tested on our vina cluster (hostnames `vina00` to `vina09`). Marks will be awarded on the basis of performance on the vina cluster and no other machines. The assignment must be coded in C. Please see the assignments page for more details. Also see the submission page for more information about how to submit and how the submission will be tested.

Milestone 1

The goal of milestone 1 is to implement the `dsm` program and the *initialisation* and *barrier* functionality of the SM library.

dsm

The program that starts the allocator and node processes is called `dsm`. It is started with command line parameters that specify a client executable to run, a number of instances of the client to start, and a set of hosts on which to start those clients. It launches the specified number of client program instances on the specified set of hosts.

The following brief usage specification of `dsm` shows exactly how the program must behave.

```
Usage: dsm [OPTION]... EXECUTABLE-FILE NODE-OPTION...

  -H HOSTFILE list of host names
  -h          this usage message
  -l LOGFILE  log each significant allocator action to LOGFILE
              (e.g., read/write fault, invalidate request)
  -n N        fork N node processes
  -v          print version information

Starts the allocator, which forks N copies (one copy if -n not given) of
EXECUTABLE-FILE.  The NODE-OPTIONs are passed as arguments to the node
processes.  The hosts on which node processes are started are given in
```

> *HOSTFILE, which defaults to `hosts'. If the file does not exist,*
> *`localhost' is used.*

In this specification `EXECUTABLE-FILE` is the client program, which must be linked against the SM library. The parameter `EXECUTABLE-FILE` will be an absolute file name, and must refer to the exact same executable on all hosts listed in the hosts file. It is up to the user of the DSM system to ensure that this condition is met. On the vina cluster, NFS ensures that all home directories are uniformly available on all nodes.

The format of the log file is not specified, however, see the sample applications for an example.

The `dsm` program starts `N` node processes running the client program and assigns each node process an identifier between `0` and `N-1`.

The `NODE-OPTION` arguments are the parameters that are passed to the client program.

The `HOSTFILE` argument names a text file that specifies the nodes on which the node processes will be started. Each line of this file contains a single hostname. The first node process is started on the first named host, the second process on the second named host, and so on. If there are more node processes than host names, the host assignment process goes back to the first host name after having started a node process on the last host name in the list. If the `-H` option is not given, the name of the hosts file defaults to `hosts`. If the file does not exist or is empty, `localhost` is used as the only default hostname.

Starting Client Programs

After `dsm` is started, it `fork()`s the node processes. Note that the `fork()`ed processes are not the client processes themselves; instead, they need to use `ssh` to start the actual clients on the designated hosts.

Allocator

After starting the clients, `dsm` takes on the role of the allocator process and continues to manage communication in the system.

Communication

All communication in the system is between the allocator and the node processes - node processes never communicate directly amongst themselves. This is very harmful for scalability, but also significantly simplifies the assignment. Synchronisation is greatly simplified when all communication goes via the central allocator, as this orders all messages in the system at the central allocator. Hence, it is easier to implement a multiple-reader/single-writer protocol correctly.

Use TCP stream sockets (`AF_INET` domain) for communication between node processes and the allocator. Do not use fixed port numbers. Pass the information required by node processes to connect to the allocator (i.e., hostname and port number) as command line arguments to the client program, and let `sm_node_init()` process these.

SM library API

The file `sm.h` defines the shared memory library API. Do **not** under any circumstances modify `sm.h`. Furthermore `sm.h` is the **only** header that you may require client programs to include. In other words do not provide any other header files in your implementation that client programs will have to include.

The identifiers of all functions and non-`static` global variables in the SM library must be prefixed with `sm_` to avoid name clashes with client code. (Otherwise, you may lose marks due to non-working test code.)

For milestone 1, you must implement `sm_node_init()`, `sm_node_exit()`, and `sm_barrier()`. Provide dummy (e.g., empty) implementations for the rest of the functions in `sm.h`.

```
int sm_node_init (int *argc, char **argv[], int *nodes, int *nid)
```

Client programs have to invoke `sm_node_init()` before calling any other function from `sm.h`. The allocator must pass the node process some information on startup (e.g., the IP address and port number on which the allocator can be contacted). It passes this information as arguments and `sm_node_init()` must extract that information from the arguments passed in `argv`.

The idea here is that when the system is invoked with, for example, `dsm -n 5 NODEPRGM ARG1 .. ARGN` then, `dsm` will pass the arguments `ARG1` to `ARGN` to all `NODEPRGM`s. However, `dsm` internally has to give some extra parameters to the node processes (IP number, port number, etc). Now, `sm_node_init()` should **remove** all these internal arguments and leave only `ARG1` to `ARGN` for the user program.

```
void sm_node_exit(void)
```

Client programs have to call `sm_node_exit()` immediately before they `exit()`.

```
void sm_barrier(void)
```

Synchronisation between the client processes is done using the `sm_barrier()` function. In order to use a barrier for synchronisation all clients must call the `sm_barrier()` function, after which they will wait until each client has executed the barrier. After the last process has executed the barrier all processes continue with their execution simultaneously. Note that all client processes must participate in each barrier, otherwise the program will deadlock. While this synchronisation primitive is rather simple, it is sufficient for small example applications.

Use the allocator process to co-ordinate barrier synchronisations - this makes them rather easy to implement.

Documentation

Document the design of your implementation and highlight any special features or shortcomings. Write this in a plain text (ASCII) file named `DESIGN` that you must include with your submission.

Milestone 2

The goal of milestone 2 is to extend the SM library and the allocator to provide a shared address space to all the nodes running the client program.

Allocator

The allocator manages the shared memory by keeping track of the location and access rights of each page. It receives and mediates client requests for pages, retrieving and distributing pages as necessary. When it receives a request for a read copy of a page, it simply retrieves a copy of the page and sends it on to the client that requested it. When it receives a request for a write copy of a page however, it must also send write-invalidate messages to all nodes that maintain a copy of that page. Note that if a node has write access to a page then no other node should have read or write access to that page.

Communication

Since client processes have to honour write-invalidate messages immediately this assignment will not work with a pure client/server architecture. While the allocator's core remains essentially an event or message loop, where it waits for requests from node processes that trigger allocation actions and messages to node processes, node processes need to be able to immediately react to unsolicited messages, such as write-invalidate requests from the allocator process.

For node processes use asynchronous socket communication together with signals in order to be notified of incoming messages when they arrive. You can achieve this by using the Posix `SIGPOLL` (aka `SIGIO`) signal for asynchronous notification of the arrival of data.

In the node processes, use `sigaction()` to register handlers for (1) catching segmentation faults due to protected or unmapped shared memory and (2) catching signals due to asynchronous communication from the allocator (e.g., to process write-invalidate messages).

See the man pages of `sigaction()` as well as of `fcntl()` (in particular, the text related to `O_ASYNC`). You may also want to take into account some further hints regarding signal handlers.

Memory management

Let the allocator control the allocation of pages in the shared memory area. The allocator can then hand the pages (or more precisely the address ranges) to the node processes on a per page basis. Each node process can internally allocate areas of arbitrary size from the pool of pages that the allocator gave to that specific node. But be careful that you can also deal with `sm_malloc()` calls that request space for objects that are larger than the size of an individual memory page.

Use `mmap()` and `mprotect()` to allocate the shared memory region and to alter the protection on individual pages in that region, respectively.

The size of the shared memory region should be 0xffff pages of memory. On Linux, this amounts to an upper limit of 256MB of shared memory, which is plenty. You can get the size of memory pages with `getpagesize()`.

In this assignment `mmap()` is used by a process to reserve a local region of memory. When you `mmap()` anonymous memory it informs the kernel that this memory is actually used in your process. After `mmap()`ing a region of memory you can use `mprotect()` to change the access permissions on that memory.

Allocate all shared memory in a continuous area starting from the same location on every node. This is done by invoking `mmap()` with the `MAP_FIXED` flag.

Determining the start address of the shared memory area is tricky.

- You can let the OS decide where to put the shared memory. To do this, the allocator should call `mmap()` with `NULL` as the first argument and *without* the `MAP_FIXED` flag. The resulting address is the start of an appropriate memory area on that node.
- Note, however, that the address returned by `mmap()` when used as above may be different on the different nodes, and hence it may not be safe to use the same address on another node (see, for example, this article for more information on the virtual memory address space on Linux). You may need to do some extra work to find an address that works on all the nodes.

SM library API

For milestone 2, you must implement `sm_malloc()` and `sm_bcast()`.

`void *sm_malloc (size_t size)`

Allocation of shared memory is done using the `sm_malloc()` function, which allocates a region of the virtual address space to use as shared memory.

No two `sm_malloc()` calls - independent of whether they are executed in the same node process or not - may return the same address. Just to make this completely clear, if `sm_malloc()` is executed by Node #1 and earlier, later, or simultaneously by Node #2, the two memory areas returned by the two calls must be disjoint! (How you achieve this is up to you - in fact it is one of the things I'd like you to think about in this assignment. But let me say that there is a simple solution, which does not require any additional communication.)

Multiple consecutive calls to `sm_malloc()` requesting memory chunks that are much smaller than the system's page size should be allocated on a single page. *But* there is no need to optimise the allocation across node processes; i.e., if two *different* node processes allocate one byte each, these two bytes may be allocated on different pages.

Note that there is no facility for freeing shared memory (it would require significant work and is largely orthogonal to the purpose of this exercise).

`void sm_bcast(void **addr, int root_nid)`

Clients exchange the addresses of allocated shared memory regions using the `sm_bcast()` function, which allows one node process to communicate the address of an object located in shared memory to all other node processes. Imagine process #1 has allocated some shared memory with `sm_malloc()`. It can now use that object, but as it is the only process knowing the address of the object, it is not yet very useful for communicating with other processes. With the help of `sm_bcast()`, process #1 can communicate the address of the shared object to all the other processes. Now all of them can access the object.

The `sm_bcast()` function also acts as a barrier, and as with the barrier function, all client processes must participate in each bcast, otherwise the program will deadlock.

The example applications show how it can be used.

Documentation

Extend the documentation from milestone 1 to discuss the design of your implementation of shared memory and highlight any special features or shortcomings. In particular focus on explaining the page request/invalidation/transfer protocol and key data structures used to implement it. You may want to use (ASCII) sequence diagrams to help illustrate this. The documentation should be in a plain text (ASCII) file named `DESIGN` that you must include with your submission.

General Requirements and Tips

Clean-up and error handling

Make sure that the allocator and node processes clean up properly. This includes `wait()`ing for terminated child processes and ensuring that node processes automatically terminate if the allocator closes the TCP stream or does not respond anymore. Failure to do so will cost style marks.

You must also ensure that your program cleans up after itself, even if it terminates due to an error condition. This implies the removal of all locally or remotely spawned processes and the removal of all created files (other than the log file). Also, make sure that you test for possible error conditions of system calls and gracefully terminate where fatal errors are diagnosed. Failure to do so will cost style marks.

Signals

The assignment requires the extensive use of signals. Make sure that system calls are either restarted when interrupted or that you handle `EINTR` properly, otherwise, your program may be "flaky". During autotesting, programs will be run multiple times on the same input. Programs must deterministically return the same results on every run (with the exception of indeterminism due to concurrency in client code and ordering of messages in the log file where appropriate).

Implementation Suggestions

The following are merely suggestions to help you. You are not required to proceed as indicated here if you prefer an alternative implementation.

- You may use `getopt()` for command line processing.
- You may want to consider these debugging tips.

The following examples show how to use a signal handler for SEGV as well as how to use `mmap()` and `mprotect()`. There is also an example of setting up sockets to generate signals when data arrives on them.

- The program `fault_addr.c` demonstrates how to catch a SEGV and how to determine the fault address in the signaa handler. This requires at least a **2.4** Linux kernel; it will not work with earlier kernels.
- The program `mmap_example.c` demonstrates the use of `mmap()`.
- The program `signal_example.c` combines the previous two examples and enables access to `mmap()`ed memory with `mprotect()` in a SEGV handler.
- The program `signal_client.c` shows how to set up asynchronous sockets to generate signals when data arrives on them. `server.c` is an example server program that can be used together with it.

Example Applications

Two example applications of the DSM system are provided separately.

Frequently Asked Questions

There is a FAQ. If you have any doubts or questions, save yourself some effort and check here before searching online, asking on the forum, or emailing.

Deadlines & Submission Procedure

This is a *team assignment.* You are expected to solve the assignment in groups of two or three. However, if you really can't work in a group you may also do it individually.

Deadlines

The submission deadlines are:

- Milestone 1: Sunday, 18 March 2018 (23:58 AEDT).
- Milestone 2: , Tuesday, 3 April 2018 (23:58 AEST).

A milestone may not be submitted before the previous milestone has been submitted.

Submission

Please follow the submission guidelines (otherwise, up to 10% of the full mark for the assignment may be lost).

Marks

The total assignment mark will be out of 40, with a maximum of 30 marks for correct code functionality, 5 marks for the code and code style, and 5 for the design documentation. While the functionality marks are spread out over the milestones, the code style and documentation marks will be based on all the code and documentation submitted over the 2 milestones.

The marks are broken down by milestone as follows:

- Milestone 1 - functionality: 7 marks
- Milestone 2 - functionality: 23 marks
- Overall code style: 5 marks
- Overall documentation: 5 marks

The penalty for late submission of assignments is 6% of the maximum possible mark per day of being late (for the milestone deadline). (Note that this is deducted from the achieved mark, not the maximum achievable mark - see the course outline page for an example.) No assignment will be accepted later than one week after the deadline.

There are harsh penalties for plagiarism (e.g., copying and teamwork outside of official assignment teams), including 0FL for the whole course. Please see the course outline page for details. Also, make sure your assignment solutions are not available to others online (for example, hosted on github.com) as this could also lead to and be regarded as plagiarism.

[1]. Under the write-invalidate policy, at most one thread is allowed write access to a given memory object. Hence, before a write may be acted on, the thread must acquire write permission for that memory object, which includes the invalidation of all other copies of the same object. If multiple copies of a memory object can exist for read access, this policy is known as multiple-reader/single-writer.