Distributed Shared Memory Implementation

Drew Haven, Justin Holmgren, Katherine Fang

Overview

We implemented a distributed shared memory system in Linux based primarily on the paper by Li and Hudak [1]. The system allows processes to share regions of memory such that it looks to the code that they're reading and writing the same portion of memory, even if they're running on different computers. In our system, we used a centralized manager to help track the pages and implemented a program based on Strassen's matrix multiplication algorithm [2] to benchmark its speed.

Implementation

Our implementation of distributed shared memory is written on top of user space read and write fault handling in Linux. We closely followed the protocol presented in [1] using a centralized manager and an acknowledgement system in order to track the different 4096-byte pages of shared memory. However, our manager process does *not* do any other processing, and is instead responsible solely for passing messages and tracking state. This state consists of a page owner and a copyset for each page of shared memory.

The page owner has the authoritative copy of that chunk of memory, and all other processes which wish to read must communicate with it in order to get a copy. In order to get a write copy, the manager facilitates in transferring ownership from the old owner to the new writer. This system does not rely on acquire / release mechanisms, but rather, any page can revoke another page's read or write access at any time.

In order for another program to use DSM, we have provided a library interface which other programs can use. The important API calls are:

- `dsm_init(id)`
- `dsm_open(addr, size)`
- `dsm_reserve(addr, size)`
- `dsm_malloc(size)`
- `dsm_free(addr, size)`
- `dsm_lock_init(addr)`

`dsm_init` simply initializes the process to hook into our DSM system. This sets up the read and write fault handlers in addition to setting up 3 ports on which it communicates. The first port is the request port on which read and write requests are forwarded from the manager. The second port is the info port. This port is used to accept responses from requests that the process has sent out. For example, if a page read faults, then the process will send a request to the central server and wait for that page's contents to show up on the info port. The last port is only used for communication about memory allocation and is used in the other 4 API calls.

`dsm_open` informs the central manager what region of memory it can use in `dsm_malloc`. In our current implementation, it is assumed that *all* client processes `dsm_open` the same region of memory.

dsm_reserve is similar to dsm_open, except that the central manager will *not* allocate these regions of memory when.

dsm_malloc asks the central manager for a chunk of contiguous memory. The manager then returns a shared address and marks that section of shared memory as in use. Finally **dsm_free** returns the pages to the pool of free pages that the manager can then use in future dsm_mallocs.

dsm_lock_init asks the central manager to help out in initializing a mutex lock initialized given an shared address. This call makes it possible to initialize locks, which then can be used in the same manner in which locks in the pthread library are used.

Our original implementation only included dsm_init and dsm_reserve. However, in writing a benchmark, we discovered that dsm_malloc is necessary as it is unreasonable to expect programmers to write programs with the knowledge of the virtual addresses of the interesting chunks of memory. dsm_lock_init was necessary in order to introduce locking, which in turn was necessary to guarantee the correctness of the benchmark and potentially any other program that runs on a DSM system.

Experiments

We have several tests to run our code. Instructions on how to run them can be found in RUNNING_CODE.txt. One of the tests we made was a benchmark test based on Strassen's matrix multiplication. We found that running it on one process (with the DSM system) took 23 seconds. Much of this time was spent setting up the memory layout and calling dsm_malloc. Running it on two processes only took 18 seconds.

Number of processes	Time spent in seconds
1	23, 22
2	18, 28

Conclusions

DSM is pretty slow. From "eyeball" profiling of print statements, it looks like a lot of time is spent dsm_malloc'ing. Our system would likely benefit from decreased network traffic. We did look at the Treadmarks paper [3] and found that we had already implemented lazy release consistency. The other speedups Treadmarks presented included multiple writers, which was not super relevant in our benchmark, as well as some tricks to get the network transfers to be faster, which would likely have improved our performance.

Citations

[1] Li, Kai, and Paul Hudak. "Memory coherence in shared virtual memory systems." ACM Transactions on Computer Systems (TOCS) 7.4 (1989): 321-359.
[2] Strassen, Volker. "Gaussian elimination is not optimal." Numerische Mathematik 13.4 (1969): 354-356.
[3] Amza, Cristiana, et al. "Treadmarks: Shared memory computing on networks of workstations." *Computer* 29.2 (1996): 18-28.