CPSC-662 Project 3

Problem 3: Distributed Shared Memory

Due date: To Be Announced

1 Overview

The objective of this machine problem is to implement a simple page-based distributed shared memory (DSM) system. The system should eventually consist of a library (plus probably one or more daemons) which can be linked into an application program and take over the management of distributed shared memory.
The application interface for the library consists of a small number of function calls that enable the user to open shared memory segments and to do some primitive synchronization. A listing of the interface (file ”dsm.h”) in the C Language is given below.

```
typedef unsigned long Size;   /* Size of a DSM segment        */
typedef void * Address;       /* Pointer to memory location   */

/* -- INITIALIZATION AND STUFF */
extern void initDSM(<...perhaps you need parameters here...>);
/* Initialize the DSM subsystem.  Must be called at the beginning of every
   program using the DSM system. */
extern void terminateDSM();
/* Is called before the application program terminates. */

/* -- MANAGEMENT OF DSM SEGMENTS */
extern Address openDSMSegment(int id, Size size, BOOLEAN creator);
/* Creates a local DSM segment with given identifier and size (in bytes).
   The third arguments specifies whether the process is the creator for
   the segment. Only one process can be creator for a particular DSM segment.
   Returns the local base address for the given segment (similar to malloc). */

/* NOTE: We omit closeDSMSegment */

/* -- BARRIER SYNCHRONIZATION */
extern void initBarrier(int bid, int n);
/* Create a barrier with identifier 'bid', for 'n' processes. Future waits
   on this barrier will be blocked until 'n' processes are waiting. */
extern int waitAtBarrier(int bid);
/* Block at barrier 'bid' until all processes have reached the barrier.
   This function returns -1 if it fails (e.g. if it is called before
   the barrier is initialized locally with waitAtBarrier). */

/* -- LOCK SYNCHRONIZATION */
extern void initLock(int ba);
```

```
/* Create a lock with identifier 'lid'. */
extern int acquireLock(int lid);
extern int releaseLock(int lid);
/* Acquire (lock) and release Lock with identifier 'id'.
   This function returns -1 if it fails (e.g. if it is called before
   the lock is initialized locally with initLock or if the lock is released without
   having been previously acquired). */
```

For simplicity, we will realize a sequentially consistent memory model. The synchronization primitives are simply for ease of programming. An example program may look as follows:

```
#include "dsm.h"
#include <some-other-stuff-perhaps>
#define ID 1
#define BA 1

int main(int argc, char * argv[]) {
    int i;
    BOOLEAN master = FALSE;
    int * shared_array;

    while ((c = getopt(argc, argv, "M")) != -1)  /* check argument string */
        switch (c) {
            case 'M' : master = TRUE;
                       break;
        }

    initDSM();
    shared_array = (int*)openDSMSegment(ID, 10000 * sizeof(int), master);
    if (master) initBarrier(BA,4); /* one master and three slave processes */

    for(;;) {
        assert(waitAtBarrier(BA) != -1);
        for(i=0; i<9999, i++)
            array[i] = i;   /* write */

        FOR(i, 0, 9999)
            if (i % 100 == 0)
                printf("array[%d] = %d\n", i, array[i]); /* read */
    }
}
```

2 Implementation of DSM

In this implementation, we rely on the memory management functions (`mprotect`, `mmap`) provided by UNIX to give us the hooks to realize distributed shared memory. In particular, these functions allow us to generate traps when particular portions of memory are accessed, which then can be caught by signal handlers and transformed into remote paging operations. The three main components of the DSM segment management at each process are (i) the creation/initialization of a new DSM segment, (ii) the signal handler that resolves accesses to invalid memory pages, and (iii) the *page stealer*, which serves requests for pages from other processes.

2.1 Creation/Initialization of DSM Segments

The `openDSMSegment` operation on a DSM segment first allocates the memory and then sets the protection bits appropriately. When it allocates the memory for the segment, it needs to make sure that the latter is aligned at page boundary. A (somewhat convoluted) way to do this is by mapping the memory location to a file, as follows (if you want to understand the meaning of the parameters, check the appropriate man pages):

```
if ((devzero_fd = open("/dev/zero", O_RDWR, 0)) < 0)  {
   perror("open devzero");
   exit(1);
}

base_addr = mmap(0, size, PROT_NONE, MAP_PRIVATE, devzero_fd, 0);

close(defzero_fd);
```

After the memory has been allocated, the protection bits are set appropriately:

```
mprotect((char*)base_addr, size, <protection>);
```

where protection is `PROT_WRITE | PROT_READ` for the creator and `PROT_NONE` for all others. The creator can now read from and write to the segment, and everybody else traps on any access (read or write) to that segment. The trap happens in form of a `SIGSEGV` (segmentation fault) signal, which can be caught by an appropriate handler. A memory mapped region is automatically unampped when the process terminates or by calling `munmap` directly. Closing the file descriptor does not unmap the region.

2.2 Signal Handler

In order to make a decision on what to do with the signal, the signal handler needs to have information about the signal, e.g. whether indeed the trap occured because of insufficient access rights, or what the accessed address was when the trap occured. The system call `signal` does not allow to pass this information. Use the call `sigaction` instead. The following piece of code illustrates how this can be done by using the `/dev/zero` file (again, check the man pages if you don't understand).

```
struct sigaction act, oact;
act.sa_sigaction = sig_SIGSEGV;
```

```
  sigemptyset(&act.sa_mask);
  act.sa_flags     = SA_SIGINFO;
  sigaction(SIGSEGV, &act, &oact);
```

The signal handler `sig_handler` can now access the information as follows:

```
  void sig_SIGSEGV(int sig, siginfo_t * info, void * dummy) {
   /* The signal code is now in info->si_code. (should be SEGV_ACCERR)    */
   /* The starting address of the page being accessed is in info->si_addr.*/
  }
```

If the signal handler detects that a shared page is being accessed for which no local copy exists, it invokes a *remote page fault*: it gets a copy of the page from remote and copies it into the appropriate location. After the copy has been installed locally, the protection bits are updated appropriately, using the `mprotect` system call. The signal handler then returns, after which the system resumes the access to the memory location, which now should succeed.

2.3 Page Stealer

Each process must deal with incoming requests for page copies and/or invalidations. Probably the simplest way to do this is by starting a *page stealer* daemon thread at DSM initialization time, i.e. in `initDSM`. (Using threads makes sharing of the memory space with the rest of the application much simpler.)
The page stealer daemon would be forked off as follows:

```
#include <thread.h>
/* Compile all source files of the program with the -D_REENTRANT option. */
/* Specify -lthreads when you link. */
void * stealer(void * dummy) {
   <...process incoming requests for pages...>
}
thread_t stealer_id;
void initDSM(...){
    ....
    if (thr_create(NULL, 0, stealer, NULL, THR_DAEMON, &stealer_id) != 0) {
        perror("initDSM: thr_create");
        exit(1);
    }
    ....
}
```

Since you have two threads running in the same address space, and they are accessing shared data, you have to determine where race conditions can occur and prevent this by appropriately using locks (look at the threads man pages for this).

2.4 Locating Valid Copies

A simple approach to locating pages is the use of a centralized *page manager*, which runs on a well-known host. Whenever a process needs to access a remote page, it asks the page manager about its whereabouts. If the page moves, the page manager needs to be informed. The page manager can be implemented as a remote program with a SunRPC interface (similar to the registry in previous MPs).

2.5 Problem

Whenever a `SIGSEGV` signal occurs because of an access to a protected page, it is difficult to determine whether the operation was a read or a write to that page. This means that we don't know whether to invalidate other copies or not. Because of this, you don't need to implement replication. Treat all shared variables as migratory. There is a single copy of the page, which always migrates to the process that accessed it last. The performance of this approach will not be great. You will get *bonus points* if you manage to solve this problem.
If you manage to solve this problem, you can keep multiple readable copies around. These copies need to be invalidated when needed. This could be done using the software messaging bus developed in the previous MP (does this tickle your ambitions?).

3 Implementing Locks

Your simple implementation of locks (barriers and simple locks) would rely on a centralized *lock manager*. Whenever a process creates a barrier or a lock, or whenever it performs an operation on a barrier or a lock, the lock manager is informed accordingly.
Note that in our simple scheme barrier locks do not get into the picture for memory consistency. The memory is kept sequentially consistent by the DSM segment management.
In order to make operations on barrier locks reasonably robust, a counter is kept with each local copy of the barrier lock. Whenever a `waitAtBarrier` operation is performed, the current value of the counter is increased and forwarded to the barrier manager, which then can compare if a process went out-of-synch.
Note that implementing the barrier manager as a SunRPC remote program may be a bad idea. Execution of a remote program is serialized, which makes it difficult to synchronize processes using RPC. A straightforward socket-based implementation probably works out better.

4 What to Hand In

The handin procedure will be the same as for previous machine problems: design, implementation, hardcopy, some performance measurements, and an analysis of the measurement results (see below for details).
The design will describe the components of the system and their interaction: the page manager, the lock manager, and the library to be linked into the application. You will

have to motivate all non-trivial design decisions. For example: why do you use RPC for a manager, why not? What is the replication approach you are taking? Why?
The running system will consist of three components: one daemon (called `dsmpd`) running the page manager, one lock manager daemon (called `dsmld`) and a test application (called `dsmtest`). The test program will be handed out separately, and will use the interface defined in file "dsm.h" listed above.
The three components are described in more detail below. If any of the components listens on a socket, the port is supposed to be well-known (i.e. feel free to hardcode it into your code – beware of collisions across groups, though!).

`dsmpd`: The page manager daemon. It processes incoming requests for page location and page transfer.

`dsmld`: The lock manager daemon. Processes incoming requests for barrier and lock creation and for operations on barriers and on locks.

`dsmtest`: A simple distributed application, on a specified number of machines. When its starts, it reads a file `dsmtesthosts`, which contains the names of all the hosts that can participate in the distributed computation. It then initializes a distributed shared memory array. After that, it "forks" off the specified number of processes on the machines defined in the file `dsmtesthosts`. The processors then compute the solution.

The program is invoked as follows:

```
dsmtest [-M] -n <nprocs> [-i <nodeid>] -P <dsmpd-host> -B <dsmbd-host>
```

where `<dsmpd-host>` is the name of the host running the page manager, `<dsmbd-host>` the name of the host running the barrier manager. The optional argument `-M` is set if the program is invoked as the master process. If this arguments is missing, the program was invoked as a subprocess by the master program. The argument `<nprocs>` specifies the number of processes (including the master!) the computation is to run on. When the master "forks" a slave process, it specifies the id (in the computation) for that process to be `<nodeid>`. You can get the source code for `dsmtest` from the Web page for this MP. Feel free to modify the source code if there is a need for it in your implementation. You will have to mark and motivate each modification in your handin, however.

The TA will test the program by issuing a sequence of commands that look similar/identical to the following ones:

```
rsh -n vanilla   dsmpd     % start page manager on vanilla
rsh -n cranberry dsmld     % start lock manager on cranberry
%run the test program on 4 processors, the master plus three listed in
%the dsmtesthosts file.
rsh -n almond    dsmtest -M -n 4 -P vanilla -B cranberry
```

The `dsmtesthosts` file would look similar to the following (feel free to put whatever machines you think is appropriate)

```
strawberry
cranberry
vanilla
sparc40
hawk
```

The measurements will consist of a number of experiments where you vary the number of processors involved, and measure latency for remote page access, number of copies/transfered pages, blocking time on barriers, etc.

5 Bonus Points

For this machine problem you can gain bonus points by implementing the DSM system at a more sophisticated level. This can be done in three steps. Before attempting Step i, make sure that you have implemented Step $i-1$ correctly!

Step 1: This is the basic level as described above. Unless your professional ambitions are limited to wanting to spend the rest of your life testing other people's Visual Basic programs, you don't want to turn in this MP at this level.

Step 2: As we stated above, the performance is really lousy if you don't know if a segmentation fault was caused by a read or a write operation. For this step, you will find a way to figure this out, and then will implement the system so that it is able to maintain multiple readable copies at any given time. (Hint: The signal handler is passed what is basically the program counter. From there, it is easy to figure out what operation was being performed when the signal happened. Peek around in the UNIX man pages to learn more about this.)

Step 3: Once you have Step 2 going, it is a piece of cake to implement release consistency. Keep in mind that we are not associating variables with locks.

Step 4: To add icing on your project, implement a *diff*-based scheme to reduce false sharing and conserve network bandwidth.