Attached is the SAS code in its current state for you to look at and the beginnings of a README. I intend to propose this as a Open Source Project, but for now please treat it a IBM internal use.

SAS stands for Shared Address Space but perhaps a better name would be Shared Persistent Heap (SPH).

What does SAS/SPH do:

It manages a "region" of the processes address space to provide:

- Sharing of data withing that region between cooperating processes
- Persistence of data within that region by providing and managing backing file space and mmaping the backing files for the process.
- Utilities for allocating/deallocating "blocks" of persistent data. Further tools for managing "blocks" as heaps, stacks, indexes...
- Locking primitives to provide synchronization of use/update between cooperating processes and threads.

Some additional definitions. For now the size and virtual address range of the region is fixed for each platform (somewhere beyond TASK_UNMAPPED_BASE and below the main stack). Blocks are allocate in power-of-2 sizes from within the region. Backing storage (files) is also allocated in in power-of-2 sizes called "segments". "Segments must be smaller in size then the "region" and usually larger then "blocks". Segments don't necessarily have anything to do with any notion of hardware segmentation, but it may be useful if the size/alignment of SAS/SPH segments match the underlying hardware.

Now a simple example of using SAS/SPH:

```
#include <sassim.h>

int main ()
{
  int rc;

  rc = SASJoinRegion();

  if (!rc)
  {
     char *block;

     block = (char*)SASBlockAlloc (4096);



     SASCleanUp();
     return 1;
  }
  return 1;
}
```

Note that the application must first "join" a region. If the region does not exist "join" will create it. By default the region is "." or the current directory. The region name can be overridden by setting the "SASSTOREPATH" environment variable. Basically the "region" name is the

path to a directory where the SAS/SPH backing files will be created.

Processes that join with the same region name will share the region and all of its (allocated) storage. All processes that share a region can allocate, deallocate, reference and update blocks with the region. Processes that use a different region name are independent and only see the region that they have joined. There no limit to that number of independent regions other then those imposed by the file system.

Once a block is allocated it is backed with file storage, and implicitly mmaped, for persistence and sharing between processes. The block will always be mapped at the same virtual address each time it is mapped and for each sharing process. This allows complex pointer based data structures to be stored, persisted, and shared without additional effort (the pointers are context independent).

A lock manager is provides so utilities and cooperating processes can synchronize their activities. Locks are "keyed" by virtual address which are normally that address of some interesting shared data structure or "utility" object.

```
#include <sassim.h>
#include <sasalloc.h>
#include <saslock.h>

int main ()
{
  int rc;

  rc = SASJoinRegion();

  if (!rc)
  {
     char *block;

     block = getSASFinder();

     while (buff = getInput())
     {
         SASLock(block, SASUserLock_WRITE);
    ...
    SASUnLock(block);
     }

     SASCleanUp();
     return 0;
  }
  return 1;
}
```

Currently the lock manager supports (shared) SASUserLock_READ and (exclusive) SASUserLock_WRITE locks. The intent is to add other lock types as needed. For example the index utility could use an "INTENT" lock (which allows multiple READ locks but is exclusive with other INTENT and WRITE lock) which is upgradeable to a WRITE lock.

Note the function getSASFinder(). The SAS runtime starts with special “anchor” block at a fixed locations. The anchor block contains the

indexes used to track free space and allocations. It also contains the "finder" pointer. The setSASFinder() allows the application to save a pointer any structure of its choice. The next time that application starts, it can use getSASFinder() to obtain the original pointer and navigation from there.

A simple pointer is sufficient to anchor a linked list or quad-tree etc, but not very user friendly. SPHDE provides a number of utility objects. Utility objects use blocks of storage provide a higher level functions. For example an application could create a SASStringBTree_t or SPHContext_t and store its pointer in the finder. For example:

```
#include <sassim.h>
#include <sasalloc.h>
#include <sphcontext.h>
#include <saslock.h>

int main ()
{
  int rc;
  SPHContext_t finderContext;

  rc = SASJoinRegion();

  if (!rc)
  {
     char *block;

     block = getSASFinder();
     if (block)
       finderContext = ( SPHContext_t)block;
     else
       { /* first time logic, only executes once.  */
         finderContext = SPHContextCreate(block__Size16M);
         setSASFinder(finderContext);
       }

     block = (char*)SASBlockAlloc (block__Size4M);

     SPHContextAddName(finderContext, "my_image_buffer", block);

     ...
     block = SPHContextFindByName(finderContext, "my_old_image");
     if(block)
       {
         SASLock(block, SASUserLock_WRITE);
         ...
         SASUnLock(block);
       }
     }

     SASCleanUp();
     return 0;
  }
  return 1;
}
```

Utilities objects like Context, Index, and StringBTree can be used to create directories or index large arrays of data structures for search. The Context is a combination of a StringBTree and a Index. It allows any object (address) to have a symbolic name (or names). It also provides the reverse mapping, from address to name(s).

These (Context, Index, and StringBTree) are implemented on (as a subclass of) the SASCompoundHeap_t utility object. A SASCompoundHeap_t is a heap of SASSimpleHeap_t heaps. The Btree nodes are SASSimpleHeap_t which are normally page size allocations. This framework implicitly manages locality of reference, minimizing the number of pages/cache lines touched during any specific search.