BeSS User Guide

Version 0.8.1 (Beta) - January 17, 1996

Alexandros Biliris
William O'Connell
Euthimios Panagos

AT&T Bell Laboratories
Murray Hill, NJ 07974

Bell Labs Storage System (BeSS).

Copyrighted by AT&T 1995. All rights reserved

This document contains proprietary information of AT&T; it is provided under a license agreement containing restrictions on use and disclosure and is also protected by copyright law. Reverse engineering of software is prohibited. If this documentation is delivered to a U.S. Government agency of the department of defense, then it is delivered with Restricted Rights and the following legend is applicable:

Restricted Rights Legend: Use, duplication, or disclosure of the programs by the government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of DFARS 252.227-7013, Rights in Technical Data and Computer Software (October 1988).

If this document is delivered to a U.S. Government Agency not within the Department of Defense, then it is delivered with “Restricted Rights,” as defined in FAR 52.227-14, Rights in Data - General, including Alternate III (June 1987).

For inquires about the BeSS project, please send email to Alex Biliris at biliris@research.att.com

or write to

AT&T Bell Labs, Room 2C-221
600 Mountain Ave.,
Murray Hill, NJ 07974

User Guide Index

BeSS Documentation

The BeSS software distribution consists of a binary release and the following documentation.

BeSS User's Guide. It describes how to install the software and how to compile and run an application. It provides examples of using the software. We suggest you to go through this document first.

BeSS Reference Manual. It describes the application programming interface to all BeSS classes, function calls, enumerators, etc. It also describes the error numbers, the BeSS server, and how to customize the system to meet application specific needs.

Readers of these documents must be familiar with the C++ programming language.

CAUTION: This is a beta release. The system as well as this document are not complete.

1. Installation

1.1 System Requirements

To install and run the BeSS binary release, you will need

1. SPARCstation running SunOS release 4.1.x.
2. One of the following compilers: Sun/AT&T CC 3.0 or Sun/AT&T CC 4.0 or GNU's g++ 2.7.
3. About 5 MBytes of free disk space.

1.2 Installing the Binary Release

The BeSS binary release is specific to a particular hardware/operating system platform and to a specific compiler and it contains everything needed to write, compile, and run BeSS applications. After you copy on your machine the appropriate compressed tar file from our ftp site, preferably into a new directory, execute the following command:

```
zcat <ftp-file-name> | tar -xvf -
```

The above command will create the binaries, include files, and the BeSS library needed for running BeSS applications. Set your path environment variable to include the `bin` directory. The following shows how to do this, assuming BeSS is installed in `/usr/bess`.

If you use csh or tcsh, place the following in your .cshrc:

```
set path = ($path /usr/bess/bin)
```

If you use Bourne or Korn shell, place the following in your .profile:

```
PATH = $PATH:/usr/bess/bin; export PATH
```

2. Overview of the Architecture

BeSS stores persistent objects in databases. Objects within a database are grouped into BeSS files. The important facility provided by files is the *cursor* mechanism through which objects within a file can be retrieved, one by one. Every persistent object belongs to exactly one BeSS file. When an object is created, the application specifies either the database or the particular file within a database the object belongs to. Objects created in a database without explicit file specification are placed by the system in a *default file* associated with the database. Objects can not be moved to other files.

BeSS files are made up of a number of variable-length *object segments* -- disk regions of physically contiguous disk blocks. For clustering purposes, BeSS provides to users direct control over placement of objects on a specific object segment within a BeSS file. Figure 1 shows the logical view of BeSS entities: a database consists of a number of BeSS files; a file consists of a number of object segments on which objects are stored.

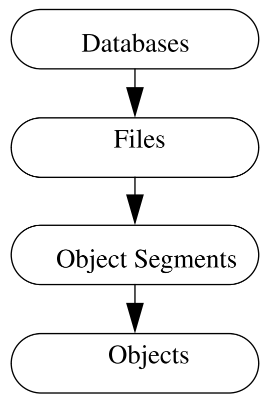

FIGURE 1. Logical view of BeSS entities

At the physical level, the database is a set of a number of storage areas, which are either UNIX files or raw disk partitions. When creating a database, the name of the first storage area of this database (i.e., the name of a UNIX file or a raw disk partition) must be provided. The database is then identified by the name of this first storage area, also referred to as the *root storage area* of this database. The root storage area is somewhat special in that it contains, besides user data, information needed by the system to manage the database (such as location of other storage areas, files, etc). Storage areas can only belong to a single database and they can be added to or removed from the space available to a database dynamically. Many BeSS files of a given database can be stored within a single storage area. A BeSS file does not span storage areas; i.e., its contents (objects) are stored in a single storage area.

As an example, Figure 2 shows a database consisting of three areas two of which are UNIX files (`/usr/alex/my_db` and `/usr/alex/indices`) and the third (`/dev/sd1d`) a raw disk partition. The name of the database is `/usr/alex/my_db` which is also the name of the root storage area. When the database is created, system files are also created in the root storage area. For example, one such system file stores information for all user files, another contains descriptors for all storage areas (their location, size, available space, format), etc.

To accommodate growth, BeSS files can be moved from one storage area to another, without affecting existing references to the objects within the files being moved. Thus, initially, the database may consists of the root area only and all user files could be created in that area. Subsequently, as files grow, new storage areas can be attached to the database and some files can be moved to new areas.

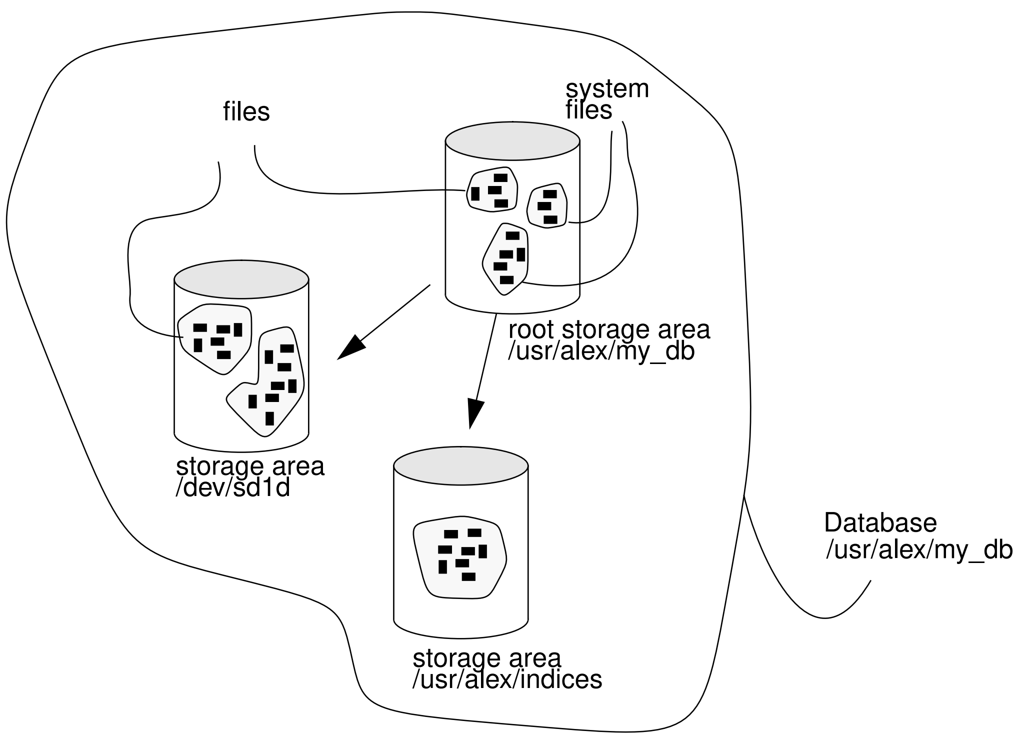

FIGURE 2. Physical view of BeSS Databases

The maximum size of a BeSS database is 128 Terabytes; this is 65,536 storage areas, up to 2 Gigabytes each. The size of an individual BeSS file can be up to 2 Gigabytes.

Storage areas, are partitioned into a number of extents, and allocation of disk segments from one of these extents is based on the binary buddy system. Storage areas that correspond to UNIX files may expand in size by one extent at a time.

2.1 Client-Server Architecture

BeSS operates on a multi-client multi-server environment. A typical BeSS network configuration is illustrated in Figure 3. There is a BeSS server running on every node in the network even if there is no database attached to the node (e.g., node 3). The server provides distributed transaction management, concurrency control and recovery. The two phase commit (2PC) protocol is employed for distributed commits and time-outs are used for distributed deadlock detection.   Two phase locking (2PL) is used for concurrency control. Moreover, client-server interaction is minimized by caching data and locks between transactions running on the same machine. Cache consistency is provided by employing the callback locking algorithm, which has been shown to have good performance over a wide range of workloads.

The server establishes a cache on the machine it's running on, processes requests of local applications to remote servers and finally processes requests from remote servers. For example, the server is responsible for fetching data requested by the local applications from the BeSS servers that own the data, acquire locks on behalf of the local applications and responds to callback requests issued by BeSS servers. Application programs will need to interact with the server running on the local machine.

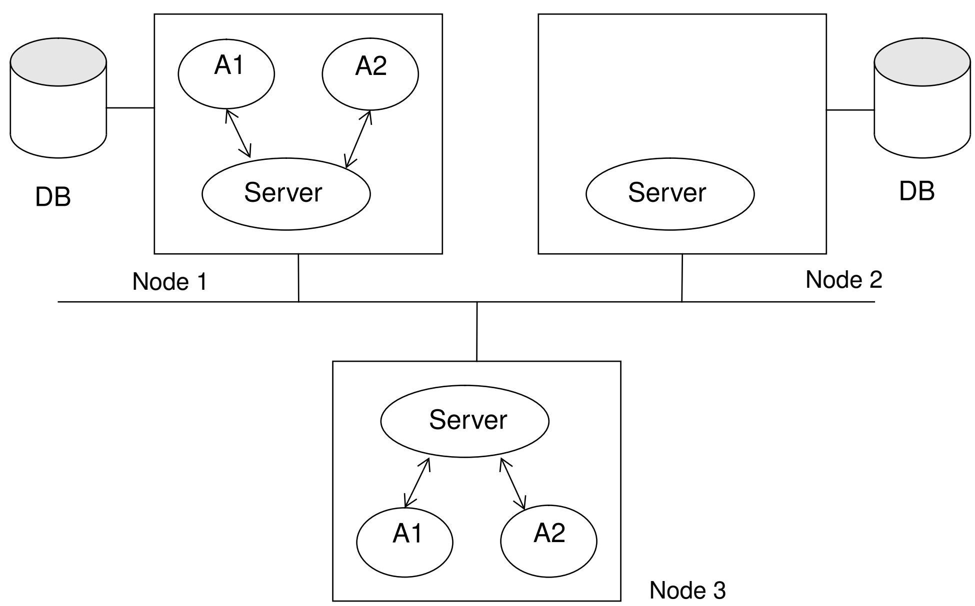

FIGURE 3. **A network of BeSS servers**

3. Getting Started

This sections describes the steps needed to compile and run a simple program.

3.1 Starting the Local BeSS Server

In order to execute BeSS applications, the BeSS server must be running on the node the application executes as well as on each node having databases needed by the application. After creating a configuration file, you can start up the server by executing the following command:

bss_server

When the server starts up successfully, the following output is displayed on the window where the server was started:

```
@(#) BeSS Client-Server Object Store -- Version x.y.z for SPARC-SUNOS
Copyright (c) 1995 AT&T all Rights Reserved.
BeSS Server: Enter comand ... (exit)
                              (debug)
                              (restart) :
```

The server accepts the above displayed commands from the standard input and occasionally produces debugging output. Thus, it is a good idea to start the server in a separate window. The only command that it is needed for the moment is "exit", which causes the server to shut down and exit. So, to shut down the server, just type `exit` on the window where the server is running.

The BeSS server uses a number of environment variables to perform its task. BeSS provides an installation program that sets up default values for all these variables so that users that do not want to customize BeSS or they are not concerned with performance need only change few things to set up their environment. Section "BeSS Customization" on page 43 of the Reference Manual describes the environment variables used by BeSS and how to change their values. If, out of curiosity, you want to see these default values, run the command:

bss_config

The output of this command is the following:

```
bss_config -- Creating Configuration File /usr/bess/.bss/bss.rc
```

indicating the file the environment variables are stored.

3.2 Compiling and Running an Application

We present below a simple makefile and two programs and we explain how to compile and run them. The source files are located in the directory `examples/example_1`. Copy them in your directory and try them out by yourself.

3.2.1 Makefile

```
BSS_HOME = /usr/bess
LIB_PATH = $(BSS_HOME)/lib
CC = CC
CFLAGS = -I$(BSS_HOME)/include

makefile: my_first_object hello_world

my_first_object: my_first_object.o
	$(CC) $(CFLAGS) my_first_object.o -L$(LIB_PATH) -o $@ -lbess

hello_world: hello_world.o
	$(CC) $(CFLAGS) hello_world.o -L$(LIB_PATH) -o $@ -lbess

clean:
	rm *.o my_first_object hello_world
```

This makefile produces two programs: `my_first_object` and `hello_world`, whose source code is given in the following two sections

The compilation command line uses the `-I` option of the compiler to add the `$(BSS_HOME)/include` directory to the list of directories the pre-processor searches for `#include` files. Application programs using BeSS must include the `bss.h` header file located in the above directory. Finally, applications must be linked with the `libbess.a` library. The BeSS library is located in `$(BSS_HOME)/lib/libbess.a`.

In order to compile and run the above programs you have to do execute the following steps:

1. Start the BeSS Server.
2. Compile the programs by using the Makefile given (just type `make`).

3 Execute the programs; run `my_first_object` and subsequently `hello_world`.

3.2.2 File my_first_object.c

```
#include <iostream.h>
#include "bss.h"

main()
{
  bss_database* db;
  int* count;

  if ( !(db = bss_database::create("/var/tmp/my_db")) )    // create a database
     cout << "Failed to create database" << endl, exit(-1);

  if ( bss_trans::begin() )                                 // begin transaction
     cout << "Failed to start a transaction" << endl, exit(-1);

  if ( ! (count = new(db) int(0)) ) {                       // create object in db
     cout << "Failed to create a persistent object" << endl;
     bss_trans::abort();
     exit(-1);
  }

  if ( db->set_object_name(count, "my_first_object") ) {   // name the object
     cout << "Failed to give a name to the object" << endl;
     bss_trans::abort();
     exit(-1);
  }

  if ( bss_trans::commit() )                                // commit transaction
     cout << "Failed to commit" << endl, exit(-1);

  db->close();
  cout << "Successfully created my first object." << endl;
}
```

This purpose of this program is to create an integer persistent object in a new database. It first creates a database in a UNIX file whose path name is provided as parameter to the static function `bss_database::create()`. This function also opens the newly created database. It then begins a transaction. Access (reading and writing) to the database can only be performed within a transaction block.

After the transaction is started, the program creates a persistent integer object in the new database. This is performed by an overloaded `operator new()` which is identical to the C++ library supplied function except that it allows for the specification of a placement argument - in the case of this example, the placement argument is the database in which the object should be created. Symmetrically, persistent objects are deleted by using an overloaded variant of the global C++ `delete` operator.

To facilitate easier object retrieval, application developers can create *named objects*, also referred to as *root objects*, whose sole purpose is to associate a name with a given persistent object. Once the association is made, a pointer to the persistent object can be retrieved at a later point by using its name. This program gives the name "my_first_object" to the newly created object by calling the function `set_object_name`. In the next program, we will show how to use this name to easily retrieve the object from the database.

Note that the program `my_first_object` should not be executed more than once for the same database because BeSS will refuse to associate a name with an object if the name is already in use.

Finally, the program calls `bss_trans::commit()` to commit the transaction and make the updates permanent, closes the databases, and prints out a success message. If any operation fails within a transaction, the transaction is aborted; i.e., the database will be in the sate it was before the transaction started.

3.2.3 File hello_world.c

```
#include <iostream.h>
#include "bss.h"

main()
{
  bss_database* db;
  int* count;

  if ( !(db = bss_database::open("/var/tmp/my_db")) )
     cout << "Failed to open database" << endl, exit(-1);

  if ( bss_trans::begin() )
     cout << "Failed to start a transaction" << endl, exit(-1);

  if ( (count = (int *) (void *) db->lookup_object("my_first_object")) == 0) {
     cout << "Failed to get the object by its name" << endl;
     bss_trans::abort();
     exit(-1);
  }

  *count += 1;
  cout << "Hello World " << *count << "-times" << endl;

  if ( bss_trans::commit() )
     cout << "Failed to commit the updates" << endl, exit(-1);

  db->close();
}
```

This program increments by one the integer object named “my_first_object” that was created in the database by the previous program. It first opens the database and then begins a transaction. The next function performs a lookup in the database to get a pointer to the integer object with name “my_first_object”. The integer is increment by one and its new value is printed out. The value of this integer will be equal to the number of times the program `hello_world` is executed.

3.3 Transactions

When multiple applications are accessing the same database, there is a potential for updates performed by one application to interfere with updates belonging to another application. To manage this interference, BeSS provides a mechanism for managing concurrent accesses to object, called *transactions*. A transaction is merely a unit of work whose boundaries are determined by the application developer. BeSS guarantees that every transaction will either complete all the operations that are within its boundaries or not complete any of them. Furthermore, transactions guarantee that updates performed to a database persist, despite system failures.

Reading and writing to the database must be within a transaction. The following BeSS operations can be executed outside a transaction block:

1. `bss::initialize()`
2. `bss::terminate()`
3. `bss_database::create()`
4. `bss_database::open()`
5. `bss_database::close()`

A transaction is started by calling the static member function `bss_trans::begin()`. The transaction is committed by calling the static member function `bss_trans::commit()`. The transaction is aborted by calling the static member function `bss_trans::abort()`. An example of using transactions is given below:

```
int success = 0;
bss_trans::begin();
...
/* Body where success is updated */
...
if ( success )
    bss_trans::commit();
else
    bss_trans::abort();
```

3.4 Error Handling

Error codes are set in the external variable `bss_err_no`. The use of `bss_err_no` is identical to `errno` in the Unix programming interface. All needed variables and functions are included in the `bss.h` header file. All BeSS functions that return an integer value (unless otherwise specified in the Reference Manual) return zero for success, non-zero for failure; the failure case is almost always -1. Functions that return pointers, they return NULL (zero) on failure. The actual error number is placed in `bss_err_no`. As in Unix, the variable `bss_err_no` is not cleared on successful calls.

The function

```
const char* bss_err_string(int no)
```

returns the corresponding character string that describes the error with error number `no`. A typical use is to call this function after an error occurs during the invocation of some BeSS function:

```
if (func1())
  cout << "errot during func1 " << bss_err_string(bss_err_no) << endl;
```

Finally, the function

```
int bss_err_trace(int b)
```

sets error tracing on or off, depending on the value of `b`. Non-zero values turn tracing on, a zero value turns tracing off. When error tracing is on, error messages will automatically be printed to the screen from each function in the current function stack.

4. Persistent Objects

While regular virtual memory pointers can be used to access a newly created persistent object or an object that has been retrieved from a database, these pointers cannot be used to “link” persistent objects together because they are invalid after the application program terminates. For this reason, BeSS provides *persistent pointers*. A persistent pointer to an object is implemented by using the parameterized BeSS class `bss_Ref`. In the following example, `spouse` is a persistent pointer to an object of type `Person`.

```
class Person {
public:
  char            name[30];       // the name of a person
  bss_Ref<Person> spouse;         // her/his spouse

  int check_on_spouse()   { return this == this->spouse->spouse; }
};
```

For all practical purposes, persistent pointers behave like ordinary C++ pointers; i.e., pointed objects are accessed using the standard C++ derefence operators. In this way, persistent object manipulation is fast and very convenient for C++ programmers.

The class `bss_Ref` is parameterized with the type `T` of the object being referenced. An instance of `bss_Ref<T>` can be used anywhere `T*` is expected. In the above example, the instantiated type `bss_Ref<Person>` behaves like a pointer of type `Person`. So, a variable of type `bss_Ref<Person>` can be used as if it were of type `Person*` as indicated in the function `check_on_spouse` in the above example. In this example, `this->spouse` (of type `bss_Ref<Person>`) is dereferenced as if it was of type `Person`.

Assume now that we want to write a function `marry_them` that assigns spouses to persons. We write this function as we would write any C++ function, with no concern about persistence:

```
void marry_them(Person* p1, Person* p2)
{
  p1->spouse = p2;
  p2->spouse = p1;
}
```

```
HLEP - CANNOT GET RID OF THIS FRAME !!!!!
```

This function can be invoked for both transient and persistent objects; the parameters passed to this function could be of type `Person*`, `bss_Ref<Person>`, or mixed as indicated in the following example.

```
Person* p;
bss_Ref<Person> q;

// initialize p and q to point to some objects

marry_them(p, q);
```

So, the important point to notice from the above example is that only pointers embedded within an object need to be declared as persistent references (i.e., of type `bss_Ref`). Variables, such as `p` and `q`, and function arguments, as `p1` and `p2` in function `marry_them`, can be declared to be regular C++ pointers even though they may point to persistent objects. The function `marry_them` updates two (potentially persistent) objects. Detection of updates, so that appropriate locks are set, is done automatically by BeSS and the user need not indicate them in some special way.

BeSS requires that the application developer registers all the types that contain persistent pointers. This is needed because BeSS "fixes" these pointers when the corresponding objects are fetched from the database. This process of fixing the pointers is known as *pointer swizzling*. The registration of a user defined type that contains persistent pointers is done by creating a *type descriptor* object, which acts like a template or signature. The following example shows how to create a type descriptor object for the class `Person`.

```
bss_database* db;
bss_Ref<bss_type_desc> type_desc;
unsigned int ptr_offset = offset_of(Person, spouse);
...
type_desc = db->create_type_descriptor("Person", 1, &ptr_offset);
```

The `offset_of()` call is a macro that takes two arguments. A class name and a data member. It returns the offset of the data member from the beginning of the structure. In the above example is used to compute the offset of the attribute `spouse` from the beginning of an object of type `Person`.

4.1 Creating Persistent Objects

There are several overloadings of the global function `operator new()` which is used to create a persistent object. The first argument in all these functions indicate where the object is stored. The second optional argument indicates the type of the object. A type is described either by its name or its type descriptor. For example, the statements below correctly create a persistent object of type `Person`, as defined in the previous section:

```
Person* p         = new(db, "Person") Person;
Person* p         = new(db, type_desc) Person;
bss_Ref<Person> p = new(db, "Person") Person;
bss_Ref<Person> p = new(db, type_desc) Person;
```

If type information is committed, the system assumes that the object created contains no references to other objects. It is OK to create an object without specifying its type, if the object does not contain references to other objects. For example, the following statement is correct:

```
int* i = new(db) int;          // OK
```

The statement

```
Person* p = new(db) Person;    // wrong
```

is not correct because although, it creates an object of type `Person`, the reference within the object will not be fixed when the object is fetched in memory and therefore derefencing the pointer `spouse` will lead to wrong results.

4.2 Deleting Persistent Objects

Persistent objects can be deleted with `operator delete()`, the same way we delete C++ transient objects. For example, the following code segments creates and deletes a persistent object:

```
Person* p = new(db, "Person") Person;
delete(p);
```

Persistent references must be explicitly casted to transient pointers in order to use `operator delete()`:

```
bss_Ref<Person> p = new(db, "Person") Person;
delete( (Person *) p);
```

In addition, the member function `delete_object()` of the `bss_Ref` class can applied to delete the object:

```
bss_Ref<Person> p = new(db, "Person") Person;
p.delete_object();
```

4.3 Example

In the following sections we present a complete example that creates a persistent linear linked list. The example consists of the following files. The first file, named `Item.h`, contains the definition of the structure of each node belonging to the list. The second file, named `schema.c`, registers the type `Item` to the database. The remaining files, named `create_list.c`, `traverse_list.c` and `remove_list.c`, contain code that creates, traverses, and removes the list, respectively.

The source files are located in the directory `examples/example_2`. Copy them in your directory and try them out by yourself.

NOTE: BETTER ISOLATE MAIN() FROM FUNCTIONS -AB, 10/22/95

4.3.1 Makefile

```
BSS_HOME = /usr/bess
LIB_PATH = $(BSS_HOME)/lib
CC       = CC
CFLAGS   = -I$(BSS_HOME)/include

make: schema create_list traverse_list remove_list

schema: schema_list.o Makefile
	$(CC) $(CFLAGS) schema_list.o -L$(LIB_PATH) -o $@ -lbess

create_list: create_list.o Makefile
	$(CC) $(CFLAGS) create_list.o -L$(LIB_PATH) -o $@ -lbess

traverse_list: traverse_list.o Makefile
	$(CC) $(CFLAGS) traverse_list.o -L$(LIB_PATH) -o $@ -lbess

remove_list: remove_list.o Makefile
	$(CC) $(CFLAGS) remove_list.o -L$(LIB_PATH) -o $@ -lbess

clean:
	rm -f *.o schema create_list traverse_list remove_list
```

4.3.2 Item.h

```
#ifndef __Item_h__
#define __Item_h__

#include <iostream.h>
#include "bss.h"

class Item {
public:
   int            item_num;
   bss_Ref<Item> next_item;

   Item(int number = 0) : item_num(number), next_item(0) {}
   ~Item(void) {}
};

#endif
```

4.3.3 File schema.c

```
#include “Item.h”

void my_exit(const char* msg)
{ bss_trans::abort(); cerr << msg << endl; exit(-1); }

main(int argc, char* argv[])
{
  bss_database* db;
  bss_Ref<bss_type_desc> type_desc;
  unsigned int ptr_offset = offset_of(Item, next_item);

  if ( argc != 2 )
     cerr << “Usage: “ << argv[0] << “ dbname” << endl, exit(-1);

  if ( (db = bss_database::create(argv[1], 0666, 1)) == (bss_database *) 0 )
     cerr << “Failed to create database “ << argv[1] << endl, exit(-1);

  if ( bss_trans::begin() )
     cerr << “Failed to start a transaction” << endl, exit(-1);

  type_desc = db->create_type_desc(“Item”, 1, &ptr_offset);

  if ( type_desc == 0 )
     my_exit(“Failed to create the type descriptor object”);

  if ( bss_trans::commit() )
     my_exit(“Failed to commit the updates”);

  db->close();
}
```

4.3.4 File create_list.c

```
#include "Item.h"

void my_exit(const char* msg)
{ bss_trans::abort(); cerr << msg << endl; exit(-1); }

main(int argc, char* argv[])
{
  bss_database* db;
  Item *item;
  int NumItems;

  if ( argc != 4 )
    cerr << "Usage: " << argv[0] << " dbname numItems listName" << endl, exit(-1);

  NumItems = atoi(argv[2]);
  if ( (db = bss_database::open(argv[1])) == (bss_database *) 0 )
    cerr << "Failed to open database " << argv[1] << endl, exit(-1);

  if ( bss_trans::begin() )
    cerr << "Failed to start a transaction" << endl, exit(-1);

  if ( (item = new(db, "Item") Item(0)) == (Item *) 0 )
    my_exit("Failed to create the first Item");

  if ( db->set_object_name(item, argv[3]) )
    my_exit("Failed to create a root object");

  for (int i = 1; i < NumItems; i++) {
    if ( (item->next_item = new(db, "Item") Item(i)) == (Item *) 0 )
        my_exit("Failed to create a new Item");
    item = item->next_item;
  }

  if ( bss_trans::commit() )
    my_exit("Failed to commit the updates");

  db->close();
}
```

4.3.5 File traverse_list.c

```
#include "Item.h"

void my_exit(const char* msg)
{ bss_trans::abort(); cerr << msg << endl; exit(-1); }

main(int argc, char *argv[])
{
  bss_database* db;
  Item* item;

  if ( argc != 3 )
     cerr << "Usage: " << argv[0] << " dbname listName" << endl, exit(-1);

  if ( (db = bss_database::open(argv[1])) == (bss_database *) 0 )
     cerr << "Failed to open database " << argv[1] << endl, exit(-1);

  if ( bss_trans::begin() )
     cerr << "Failed to start a transaction" << endl, exit(-1);

  item = (Item *) (void *) db->lookup_object(argv[2]);
  while ( item != (Item *) 0 ) {
     cout << "Item with number " << item->item_num << endl;
     item = item->next_item;
  }

  if ( bss_trans::commit() )
     my_exit("Failed to commit the transaction");

  db->close();
}
```

4.3.6 File remove_list.c

```
#include "Item.h"

void my_exit(const char* msg)
{ bss_trans::abort(); cerr << msg << endl; exit(-1); }

main(int argc, char *argv[])
{
  bss_database* db;
  Item *item, *tmp;

  if ( argc != 3 )
     cerr << "Usage: " << argv[0] << " dbname listName" << endl, exit(-1);

  if ( (db = bss_database::open(argv[1])) == (bss_database *) 0 )
     cerr << "Failed to open database " << argv[1] << endl, exit(-1);

  if ( bss_trans::begin() )
     cerr << "Failed to start a transaction" << endl, exit(-1);

  item = (Item *) (void *) db->lookup_object(argv[2]);
  while ( item != (Item *) 0 ) {
     tmp = item->next_item;
     cout << "removing item with number "<< item->item_num << endl;
     delete(item);
     item = tmp;
  }

  if ( bss_trans::commit() )
     my_exit("Failed to commit the transaction");

  db->close();
}
```

5. BeSS Databases

Each BeSS database consists of one root storage area plus one or more optional storage areas. All system structures with respect to the database are kept in the root area. The class `bss_database` provides a handle (transient structure) to manipulate BeSS databases. The following static function of this class shows all the options available to users when creating a database:

```
static bss_database* create(const char* dbname, int mode=0664, int overwrite=0)
```

It creates and opens the database whose name is specified in `dbname`. The name can be either a unix file or disk raw partition. The `dbname` argument must follow the format below, where path name must be a unix path name relative to the server on which the database is created, not the client:

```
[ <hostname>: ] <unix path name>
```

When the hostname is not supplied, the unix path name may be an absolute path name or a relative path name with respect to the directory the application is running in. When the `hostname` is supplied, the unix path name must be an absolute path name with respect to the host machine.

If a database is mounted over NFS, `hostname` is not necessary. BeSS determines if the file is physically mounted over NFS, and if so, determines its machine and absolute path name where it really resides. Internally, no matter what format is provided, BeSS stores the database name as a combination of host machine and absolute path names.

The remaining arguments are used in the following way.

`mode`	the permissions given to the database when it is created, if it is a unix file.
`overwrite`	if true and the database already exists, it will be truncated.

The `mode` field applies not only to the root area, but all subsequent storage areas created in this database.

For example, the following piece of code creates the database named `phones` on machine `allegra`. The first example illustrates opening the database as a file, the second as a raw disk partition. If the database exists, it will fail since we are not providing an overwrite flag.

```
        // database as a Unix file
bss_database *db = bss_database::create("allegra:/usr/bss/people", 0666);

        // database as raw disk partition
bss_database *db = bss_database::open("allegra:/dev/sd1d", 0, 0644);
```

To open a database, its name must be provided in the following static function:

```
static bss_database* open(const char* dbname, int read_only = 0, int mode = 0)
```

The function opens a database specified by `dbname`. The `dbname` argument has the same format as the `bss_database::create()` interface. The remaining arguments are used in the following way.

`read_only`	if true, the database is opened read only.
`mode`	if true, and the database does not already exist, it will be created with value of mode.

For example, the following code opens a database called `people` (a Unix file) on machine allegra.

```
bss_database* db = bss_database::open("allegra:/usr/bess/people", 0, 0644);
```

If the application runs on the same directory the database resides, the `open()` can be called as follows:

```
bss_database* db = bss_database::open("people", 0, 0644);
```

The user is only required to open the root area of the database (via `bss_database::open`), subsequent storage areas will be opened automatically by the system when they are first referenced.

5.1 Named (Root) Objects

Any BeSS object can be named for easy retrieval. The member function of the `bss_database` class

```
int set_object_name(const bss_ref& r, const char* name)
```

establishes an association between the given name `name` and the object referenced by `r`. If the object is already named, it will be renamed with `name`. As an example, the code below creates an integer of value 100,000 in a database described by `db` and gives the name "highest_salary" to the object:

```
bss_database* db;        // assume opened
int* hs = new(db) int(100000);
db->set_object_name(hs, "highest_salary");
```

The object can later be retrieved by its name as follows:

```
int* hs = (int *) (void *) db->lookup_object("highest_salary");
```

If no object with such name exists in the database, the null reference is returned. We can explicitly query the database for an object's name:

```
cout << "Name of object is " << db->get_object_name(hs) << "." << endl;
```

The function returns the name of the object passed as parameter, or NULL if no name is associated with the object.

We may change the name "highest_salary" to "highest salary":

```
db->rename_object("highest_salary", "highest salary");
```

Finally, we may remove the name associated with the object (but not the object itself) as follows:

```
db->remove_object_name(hs);
```

Notice that as we mentioned before, the same functions can be applied if variable `hs` was declared to be of type `bss_Ref<int>` instead of `int,` as indicated below for completeness:

```
bss_Ref<int> hs = new(db) int(100000);
db->set_object_name(hs, "highest_salary");
cout << "Name of object is " << db->get_object_name(hs) << "." << endl;
hs = db->lookup_object("highest_salary");
db->remove_object_name(hs);
```

5.2 Type Descriptors

Type descriptors for a database `db` are stored in a system file that can be retrieved as follows:

```
bss_file* t_file = db->type_file();
```

This file can then be scanned, as described in section “File Cursors” on page 24, for the retrieval of all types known to a database. A pointer to a particular type descriptor can be retrieved by supplying the type’s name:

```
bss_type_desc* person_t = db->get_type_desc("Person");
```

The class `bss_type_desc` provides member functions to get information about a type. For example, the number of references with the type pointed by `person_t` and the locations of those references are retrieved as follows:

```
person_t->no_refs();
person_t->refs();
```

The last function returns a pointer to an array of integers that contain the offsets. Finally, the following function returns the name of the type:

```
person_t->name();
```

5.3 Storage Areas

To attach (and format) a storage area to a database, the name of the area has to be specified:

```
bss_database* db;      // assume opened
bss_area* area = db->create_area("/usr/bess/area1");
```

The area name’s construction is of a similarly to the database name except the hostname should not be supplied because all storage areas associated with the same database are always on the same machine. The area being attached can be a raw disk partition or unix file name. Raw partitions are formatted only while Unix files must be both created and formatted. If a Unix file name is provided, and it already exists, it will be truncated.

The names of storage areas attached to a database are stored within the database’s root area and with relative path names with respect to the root area itself. This allows users to move databases on file systems easier. For example, if the database root area is `/usr/bess/var1/db_file`, and the new storage area attached to the database is `/usr/bess/var2/area1`, then the relative name `../var2/area1` is stored internally.

To get a handle to a storage area, we need to know its name:

```
bss_area* area = db->get_area("/usr/bess/area1");
```

Having a handle to an area, allows us to get information about the area by applying member functions of the class `bss_area`. For instance, the function calls

```
area->total_free_units();
area->n_extents();
area->extent_size();
```

return, respectively, the number of free units, the number of extents, and the size (in bytes) of each extent in `area`.

The format of a storage area, except the root area, can be specified by the user when the area is attached to a database. This is just an option because there are already default values set be the system for formatting an area. The functions that get and set the default configuration parameters are static functions of class `bss`. To get the current default values apply the following function:

```
bss::get_area_conf(n_extents, extent_size, is_expandable, all_unit_size);
```

It sets in the corresponding parameters the number of extents, the extent size, wether it is expandable, and the allocation unit size. These values will be in effect when creating an area, until the user change them by specifying the default values as indicated below:

```
bss::get_area_conf(n_extents, extent_size, is_expandable, all_unit_size);
```

6. BeSS Files

BeSS files serve as a mechanism to gather related objects together. They provide facilities for sequencing through the objects they contain. Files may contain only objects and an object is a member of exactly one file. Applications may create one or more BeSS files and/or choose to use the system default file for storing objects. Files are created in a specific database. Assume the following type definition:

```
class Person {
public:
  Person(unsigned n = 0) {ssn = n; name[0] = '\0'; spouce = (Person *) 0;}
  unsigned ssn;
  char    name[20];
  bss_Ref<Person> spouce;
};
```

The following segment of code illustrates how to create a file and create an object in it.

```
bss_database* db;              // assume database opened
bss_file* myfile;
Person* p;
...
if ( (myfile = db->create_file("my_first_file")) == (bss_file*) 0 )
    // error handling

if ( (p = new(myfile, "Person") Person) == (Person *) 0)
    // error handling
...
```

In this example, the application creates a file called `my_first_file` and then creates an object in it. Applications do not have to name files, but in doing so, it helps in re-acquiring their handles at a later point. Alternatively, a file can be accessed by supplying its id, an unsigned integer, as returned from the member function `bss_file::id()`:

```
bss_uint_32 myfile_id = myfile->id();
```

A new file is always created in the root storage area of a database. The file can then be moved from one area to another; e.g., the code below moves the file to some storage area named /usr/tom/a2:

```
myfile->move("/usr/tom/a2");
```

The new area the file is moving to must have been previously attached to the database the current area belongs to; in other words, a file cannot be moved to another database. The overloaded unction `move` takes as argument either the name of the area or a descriptor for the area. The code below finds the database the file belongs to and then moves the file to the root storage area of the database:

```
bss_database* dbase = myfile->database_of();
myfile->move(dbase->root_area());
```

When a database is created, the system automatically creates a default file that is will use when inserting objects that are not explicitly placed into other BeSS files or segments. The application does not have to open this default file, it is opened automatically when opening the database. The handle for the default file is stored in the database transient object, `db->default_file()`. Thus, the following two statements are equivalent in that both create an object in the database's default file:

```
p = new(db, "Person") Person;
p = new(db->default_file(), "Person") Person;
```

The static function `bss_file::of()` returns the file of a persistent object. Thus, the following statement creates an object in the same file the object referenced by `q` belongs:

```
bss_file* f = bss_file::of(q);
Person* p = new(f, "Person") Person;
```

File handles are not valid across outer-most transactional boundaries. What this means is that a file handle is not valid once you have committed a transaction. This does not include the case where an application has nested transaction begins and commits one of the nested one (not the outer-most transaction).

The following statements acquires a file handle for a previously created file; the first does so by file name and the second by file id:

```
myfile = db->get_file("my_first_file");
myfile = db->get_file(myfile_id);
```

A number of functions provide detail information about a file. For example, `n_objs()` returns the number of objects in the file, `n_obj_segs()` returns the number of object segments, `size()` returns the number of bytes occupied by the file.

The next statement clears the file, i.e., it destroys all objects in it, and the subsequent one destroys all objects in the file and the file itself.

```
myfile->clear();
myfile->destroy();
```

6.1 File Cursors

A file cursor allows retrieval of objects contained with in a file. The retrieval of objects is performed one object at a time and in an order that is arbitrary but stable across traversals provided no objects are created or deleted and no reorganization is performed. A cursor is moved by applying two functions: `first()`, which positions the cursor to point to the first object in the file, and `next()`, which makes the cursor advance to the next object within the file. It is also possible to position the cursor at a specific object within the file using the `set()` member function. A cursor is NULL when it points beyond the of file.

A reference to the current object pointed by the cursor can be retrieved by calling `current()`. An overloaded version of this function, in addition to returning the object at which the cursor is positioned, it accepts an argument of type `bss_type_desc&` which is set to reference the type of the object at which the cursor is currently positioned. This is useful in identifying the type of an object when objects of different types are inserted within a file.

The following code creates a file, it populates it with `Person` objects, as defined above, and then scans the file to print the ssn field of all objects in it.

```
bss_database* db;                // assume database opened
bss_file* myfile;
bss_file_cursor* cursor;
bss_Ref<Person> p;

if ( (myfile = db->create_file()) == (bss_file*) 0 )
    // error handling

for (int i = 0; i < 100; i++)
  p = new(myfile, "Person") Person(i);

cursor = myfile->cursor();
cursor->first();
while (! cursor->is_null()) {
  p = cursor->current();
  cout << "Person's ssn is " << p->ssn << endl;
  cursor->next();
}
```

6.2 Clustering - Object Segments

As mentioned above, when you create a new object, you indicate where this object is stored by specifying either a database or a particular file. In this section we discuss how to create objects at specific segments within a file. (Recall that at the physical level, a BeSS file consists of a number of object segments on which objects are actually stored.) You only need to make use of this facility if you want to store two or more objects physically close to each other to optimize the cost of data transfers.

Clustering can be done in a number of ways. One way is to direct the system to place an object `p` on the same segment where another object `q` is stored. This is done when the object `p` is created. We first retrieve the object segment of the existing object:

```
bss_obj_seg* oseg = bss_obj_seg::of(q);
```

The static function `bss_obj_seg::of()` retrieves the segment of the persistent object referenced by `q`, which is used as a placement argument when creating object `p`:

```
p = new(oseg, "Person") Person;
```

For example, the code below creates a `Person` object `p` in file `myfile` and his/her spouse on the same segment `p` is stored:

```
p         = new(myfile, "Person") Person(1);
p->spouse = new(bss_obj_seg::of(p), "Person") Person(2);
```

Alternative we may first explicitly create an object segment within a file:

```
bss_obj_seg* oseg = myfile->create_obj_seg(8000);
```

or, in a database (as opposed to a specific file):

```
bss_obj_seg* oseg = db->create_obj_seg(8000);
```

which results in creating the segment in the default file of the database pointed by `db`.

When creating an object segment, the size of the segment (in bytes) must be provided. The object segment size has an upper limit of 64K bytes. The segments are always created on page boundaries, thus internally the size is rounded up to the smallest multiple of page size that is greater or equal to the specified segment size. For the example code above, the effective size of the new object segment is 8196 bytes.

After the object segment is created, you may then explicitly create the two objects on the new segment:

```
p         = new(oseg, "Person") Person(1);
p->spouce = new(oseg, "Person") Person(2);
```

The class `bss_obj_seg` provides a number of functions that deal with segments. For example, the functions

```
bss_database* db = oseg->database_of();
unsigned file_id = oseg->file_id_of();
bss_file* myfile = oseg->file_of();
```

return the database, the file id, and the file, respectively, the segment `oseg` belongs to.

7. *Indexing*

Indices are heavily used in database systems to accelerate retrieval of objects based on some key value. An index stores (key, value) pairs. The value part of a pair stored in the index may also be the address of a persistent object. The interface for manipulating indices in BeSS is provided in the class `bss_Index`:

```
template <class T> class bss_Index
```

The class is parameterized with the type `T` of the key used in the index. `T` can be any atomic built-in C++ type or a user-defined type.

7.1 Index Types

An index can be defined to be *unordered* (implemented as hash) or *ordered* (implemented as B-tree). Either one can be defined to allow or disallow *duplicate* keys. For an ordered or unordered index that disallows duplicate keys, the index can be defined to signal *error* on an attempt to insert a duplicate key or simply ignore the second insertion. The above options are specified at index creation time by means of ORing values from the following enumerators:

```
bss_index::unordered
bss_index::ordered
bss_index::no_duplicates
bss_index::allow_duplicates
bss_index::err_duplicates
```

If neither `unordered` or `ordered` is set, or if both are set, the index will be unordered. If neither `no_duplicates` or `allow_duplicates` is set, or both are set, the index will disallow duplicates. If the index does not allow duplicates and `err_duplicate` is set, an error will be returned on an attempt to insert in the index a key that already exists and the insertion will fail. If the index does not allow duplicates and `err_duplicate` is not set, no error will be signaled on an attempt to insert a duplicate key; the operation will simply be ignored. The `err_duplicate` flag has no effect when the index allows duplicates. A proper combination of these flags allows applications concerned with duplicate keys to go ahead and directly attempt to insert a key in the index without first having to perform an index lookup for this key.

7.2 Specifying Default Rank and Hash Functions for Specific Types

Before an index for some type `T` can be created, the application has to define a rank and a hash function for values of this type. BeSS provides a macro to designate a default hash and rank user-defined function for values of some particular type. It is important to realize that you must use this macro exactly once to define these two functions for a type before an index that uses values of this type is used. The macro call has the form:

```
bss_define_rank_hash(TY,RF,HF)
```

where, `TY` is a type name, `RF` is the rank function that should be used for values of `TY`, and `HF` is the hash function that should be used for values of `TY`. Function `RF` and `HF` must be of type `BSS_HASH_FUNC and BSS_RANK_FUNC`, respectively, as defined below:

```
typedef unsigned int (* BSS_HASH_FUNC) (const void *t);          // hash function
typedef int (* BSS_RANK_FUNC) (const void* t1, const void* t2);   // rank function
```

That is, the hash function must take the address of the value of some type `T` and return an unsigned integer. Notice that although the function should be written for a value of a specific type `T`, the argument used in the function must be declared to be of type `const void*`. The rank function must take the addresses of two values of the same type `T` and return a number less than zero, zero, or greater than zero indicating `t1` is less than, equal to, or greater than `t2`. Again, although the function should be written for two values of a specific type T, the two arguments used in the function must be declared to be of type `const void*`.

The following example illustrates the usage of this macro when we want to use the type `my_type`, defined below, as key of an index.

```
typedef struct my_type {
  int a;
  int b;
} my_type;

int my_type_rank(const void* v1, const void* v2)
{
  return ((my_type *) v1)->a < ((my_type *) v2)->a ? -1 :
         ( ((my_type *) v1)->a == ((my_type *) v2)->a  ? 0 : 1);
}

unsigned int my_type_hash(const void* v)
{
  return ((my_type *) v)->a ^ ((my_type *) v)->b;
}

bss_define_rank_hash(my_type, my_type_rank, my_type_hash);

main()
{
  // ...
}
```

The first step is to define the default rank and hash functions for this type. Then, the macro `bss_define_rank_hash` registers them with BeSS. We emphasize that this macro must be invoked even if the application plans to use different hash and rank functions every time an index with `my_type` values is accessed. In this case, where there is no need to define some kind of default hash and rank functions, the macro could be invoked as follows:

```
bss_define_rank_hash( my_type, 0, 0 );
```

7.3 How to Create and Open Indices

After the rank and hash function for a type `T` have been defined, the static function of `bss_Index` class:

```
create(bss_area* a, unsigned vsize, unsigned tsize, int options)
```

creates an index in the storage area specified by `a` and returns a pointer to a transient structure of type `bss_index<T>` representing the index. The `vsize` is the size in bytes of the value stored in the index and it must be zero or multiple of the word size (4 bytes, on most machines); `tsize` is the size in bytes of some tag the user wants to associate with the index. If no user-specified information needs to be stored in the index, `tsize` value should be zero. Finally, `options` specifies the type of the index being created and its value is made up by ORing values from the enumerators defined at the beginning of this section.

For example:

```
bss_database* db;   // assume is opened;
bss_Index<my_type>* i;
int opt = bss_index::ordered | bss_index::allow_duplicates;
i = bss_Index<my_type>::create(db->root(), sizeof(int), 0, opt);
if (i == 0)
    cout << "Failed to create index" << endl;
```

creates an index in the root area of a database `db`. The key is of type `my_type`, and values are integers. The index is ordered and it allows duplicate keys to be inserted in the index. Finally, the index has no user-defined tag.

Every index has an id whose value is of type `bss_index_id`. It ca be retrieved as follows:

```
bss_index_id id = i->id();
```

You will need this id to open the index later on :

```
bss_Index<my_type>* i = bss_Index<my_type>::open(db, id);
```

7.4 Index Cursors

To be completed. Please see the corresponding section in the Reference Manual.

8. *Large Objects*

To be completed. Please see the corresponding section in the Reference Manual.

8.1 Understanding Large Objects

8.2 How to Create and Open Large Objects

8.3 File-like Operations

8.4 Memory-like Operations

8.5 Downloading/Uploading BeSS Large Objects from/to Unix Files

8.6 Asynchronous (non-blocking) Operations

8.7 Prefetching