LIBRARY IMPLEMENTATION

Introduction

The ABI stipulates that the code for a common shared library must be linked to the address space of every program. The library code must be linked to logical page 0 and logical page 1 of each program. When an ExpL source program is compiled by an ExpL compiler, the compiler generates code containing calls to the library assuming the library functions will be "there" in the address space when the program is eventually loaded for execution. It is the responsibility of the OS loader to do the actual loading of the library. (In technical jargon, the library is said to be *linked at compile time* and *loaded at run time*. Unfortunately, there is lack of agreement on terminology in these matters).

All calls to the library will be a CALL to logical address 0 of the address space. The library code expects a function code and three arguments to have been pushed into the stack before the call by the application, as described here.

To implement the library, you must write code to implement the six library functions – read, write, exit, Initialize, Alloc and Free. Among these, read, write and exit just involves calling the corresponding system calls using the low-level system call interface described here. The code for the remaining three functions must be implemented in the library.

Initialize, Alloc and Free are *heap management functions*. Each application program has a heap memory region which is attached to logical pages 2 and 3 of the address space. The heap management functions support allocation and de-allocation of dynamic memory. The *Initialize* function initializes the heap data structures. The library must contain code to allocate and de-allocate memory when requested by the application. In the case of an allocation request, the address of the first memory address of the allocated memory must be returned.

Various dynamic memory allocation algorithms exist. A very simple allocation policy would be to always allocate fixed sized memory blocks. The following document discusses this simple allocation scheme and a more complex variable block allocation method called the Buddy System Allocation scheme. In the project, you will be primarily implementing a fixed size allocator. One of the exercises in the roadmap asks you to implement the buddy system allocator.

The advantage of using a library for dynamic memory allocation is that this common code can be loaded by the OS during boot time at some memory and can be attached to the address space of every application, saving memory space.

Note: The library implementation outline given below assumes that the application code will modify only memory addresses acquired using *Alloc*. If the application modifies heap region that was not allocated to it, the data structures set up by the library functions may get corrupted and the library functions may fail to work properly.

Illustration

The following gives an outline of the library implementation.

The ABI stipulates that all the library function calls are invoked using CALL 0 instruction. The library differentiates the calls using function code, pushed by the user program on to the stack, before the CALL 0 instruction. The library should read the function code from the stack and jump to the starting address of the corresponding function logic.

The following snippet gives the overview of the design of the library.

```
//get function code from stack
....

MOV R0, <function code>
MOV R1, R0
EQ R1, "Heapset"
JNZ R1, <starting address of Initialize>
MOV R1, R0
EQ R1, "Alloc"
JNZ R1, <starting address of Alloc>
....

// code for Initialize
```

```
....

RET
// code for Alloc
....

RET
....
```

As shown above, the function code that was pushed by the application program is used to direct the control of the program to the corresponding function logic. In case of Alloc(), Free() and Initialize() calls, the library contains the logic whereas in case of Read() and Write() system calls the library should invoke the kernel using low level system call interface. The appropriate arguments required for the system call can be obtained from the user stack.

A detailed explanation on the usage of library interface can be found here.

Note: As the library is directly loaded by the ExpOS loader, the code should be free of labels and all the jump instructions should have the target memory addresses.