EXPL NITC

Home About Roadmap Documentation

XSM EXECUTION ENVIRONMENT TUTORIAL

Pre-requisite reading:

1. A quick reading of the ExpOS ABI documentation for XSM machine. You may not understand the whole content now. The details will be explained as you read ahead.

Learning Objectives :

At the end of this tutorial you will be able to generate XEXE executable files containing XSM assembly language programs which can be run on the XSM simulator given to you. Along the way, you will learn how to use the system call interface and the library interface of the underlying Operating system to handle console input and output.

This tutorial helps you to gain a basic understanding of the execution environment provided by the XSM simulator. The compiler you design for the ExpL language is supposed to generate target XSM machine code that runs on the XSM Simulator provided to you. However, the bare machine cannot directly run the target code. The operating system (OS) that runs on top of the machine is the actual software that sets up an execution enviornment necessary for running the target code. Hence, the compiler's obligation is to generate target code that is understandable to the operating system.

Consequently, there must be a document provided by the OS implementation to the compiler designer that explains the interface to the operating system that the compiler must adhere to. This document is called the Application Binary Interface or ABI.

The XSM simulator given to you is actually much more than a bare XSM hardware simulator. It has the capacity to understand the ABI for the ExpOS operating System for the XSM machine. This ABI expects that the compiler generates the target file in a format called the XEXE executable format.

Your compiler simply needs to generate an executable file following the XEXE executable format and store it on the local machine. When you use the XSM simulator to run the program, the following actions take place.

1. The script that runs the simulator transfers the file from your local machine's disk to the XSM machine's (simulated) hard disk.
2. It then boots up the operating system (the OS is already preloaded in the simulator's hard disk). The bootstrap loader starts in the kernel mode and sets up a user process in memory, allocating an address space. Then, it transfers the executable file from the hard disk to the code region of this memory. Page tables are also set up to run the process in user mode.
3. Finally, the simulator sets the instruction pointer (IP) to the address specified in the entry-point field of the header of the executable file and control transfers to this instruction, resulting in execution of the loaded program starting with the instruction specified by the entry point. The machine also switches from the kernel mode to the user mode. (Technically, the OS code pushes the entry point address on to the program's stack and executes the IRET instruction resulting in transfer of control in user mode to the specified memory address. These details are not relevant for your work and are noted just for the sake of information.)

e start now by generating a small XEXE executable file containing an XSM ogram to find the sum of two numbers and store the value in a register. The lue of the register will have to be inspected in order to view the ouput. This possible by executing the XSM simulator in debug mode. Later we will see w console input and output are handled.

---

periment I : Adding two numbers using registers.

s noted above, executable programs must be designed in such a way that it ust be possible for the file to be loaded and executed by the underlying perating system. When a program is loaded into memory by the OS, the OS pically assigns a virtual address space (or simply an address space). In the esent case, the address space of a program starts at address 0 of the emory and ends at address 5119. This means that while designing the target ogram, you may assume that this is the total computer memory accessible the program.

e compiler typically divides this memory into various regions – namely the de region, (often called the text region), data region, stack region, heap gion etc. The ABI specifies the starting address and ending address of each gion in the address space. This is specified HERE. The ABI specifies that the mpiler must divide the memory into four regions – library, code, stack and ap. (there is no separate data region – instead the stack region must be ed for this as well).

r our immediete requirements, the important region is the code region. e code region contains two parts – a header (addresses 2048 to 2055) and de (address 2056-4095). The target code will be loaded into this region of e memory when the OS (simulator) loads the program for execution. etails will be described soon.

ıe XEXE executable format stipulates that the first 8 words of an executable e must be a header. The rest of the file must contain assembly language ogram to be executed. The loader actually will simply copy the contents of e executable file into the region of memory between addresses 2048 and)95. Consequently, the header will be loaded between 2048-2055 and the st of the file (containing the XSM instructions) into addresses 2056 to 4095. ıe contents of the file will be copied to the memory in the order in which ey appear in the file.

nce each XSM assembly instruction requires two words of memory storage, e first instruction will be loaded into address 2056 and 2057, the second in)58 and 2059 and so forth. Since the code region of the memory ends at)95, the maximum number of instructions possible in a program is limited 1020. This is the limit set by the particular OS platform used in this periment.

ne important field of the header is its second entry – the entry point. The ader initializes the Instruction Pointer (IP) register of the XSM machine to is value. Thus, if entry point is 2064, the simulator the XSM machine mulator will start program execution by fetching the instruction stored at is address. The ABI stipulates that the value of the first field called the magic ımber must be set to zero. The setting of other fields in the header are not levant for this experiment.

ote: When the program is actually loaded for execution by the OS, the ıysical addresses in the actual memory to which the executable program is aded will be different from the virtual addresses set by the compiler. Such location requires architecture support. In the XSM machine, the support is rough paging. However address translation is a concern of the OS, not of e compiler. Hence we will not pursue this matter here. However, for those terested, details on XSM paging scheme is given HERE.

ith this, we complete the background needed to complete the present periment. We now proceed to the implementation.

ıplementation :

ı XSM assembly language program to find the sum of two numbers could ork as the following:

```
1. Store the first number in a register – say R0.
2. Store the second number in a register – say R1.
3. ADD R0, R1.
```

ıe result will be stored in R0.
› generate code for the above tasks and write it into a target_file, you must rite code as:

```
fprintf(target_file, "BRKP");
```

```
fprintf(target_file, "MOV R0, 3\n");
fprintf(target_file, "MOV R1, 2\n");
fprintf(target_file, "ADD R0, R1\n");
```

ıe target_file will look as shown below.

```
BRKP
MOV R0, 3
MOV R1, 2
ADD R0, R1
```

target_file hosted with ❤ by **GitHub** **view raw**

owever, the header must be written into the first eight words of the target e before writing out the instructions. You must reserve the first eight words the executable file for the header before writing code into the file. Now, t the entry point field to the first instruction to be executed. If the code is ritten immedietely after the header, the first instruction will be loaded to emory address 2056 and 2057 (see memory model). Hence, you must set e header as:

```
fprintf(target_file, " %d\n %d\n %d\n %d\n %d\n %d\n
%d\n %d\n ",0,2056,0,0,0,0,0,0);
```

```
0
2056
0
0
0
0
0
0
BRKP
MOV R0, 3
MOV R1, 2
ADD R0, R1
```

target_file after adding header hosted with ❤ by **GitHub** **view raw**

ıe above code essentially sets the first field – magic number - to 0, the cond field - entry point - to 2056 and other fields to 0.

ow, to run the executable file, you must use the XSM simulator. The mulator usage commands are specified here. You must read the above link efore proceeding further.

ıe simulator expects a library file by the name library.lib together with the XE executable file to be supplied as a command line argument. You will arn more about the library in later in this documentation. For now create a e library.lib with just one instruction in the XSM simulator folder.

```
RET
```

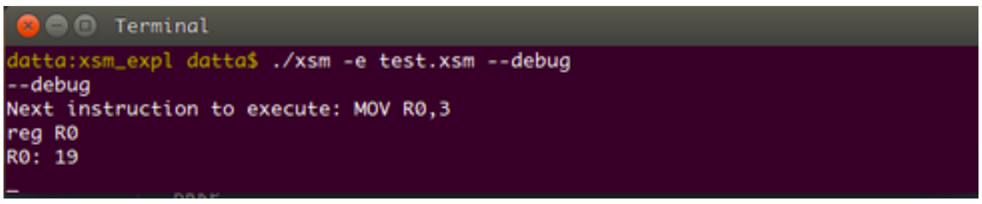
library.lib hosted with ❤ by **GitHub** **view raw**

ow we will try to execute the target_file in the debug mode.

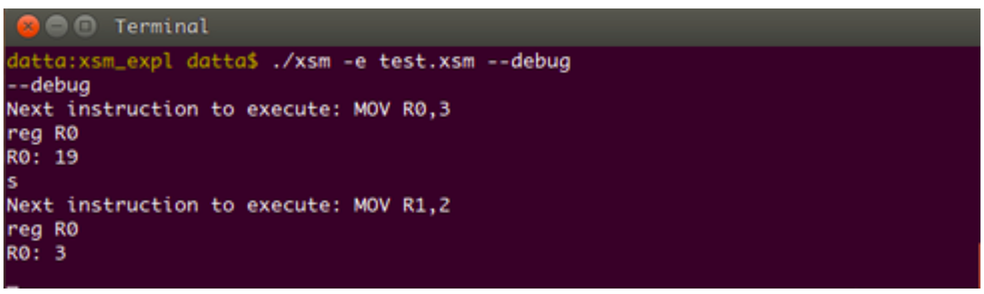
1. Open terminal and navigate to the simulator folder.
2. Type "./xsm -l library.lib -e < path to target_file.xsm > --debug".

ote: Path to the xsm file need to be relative path.

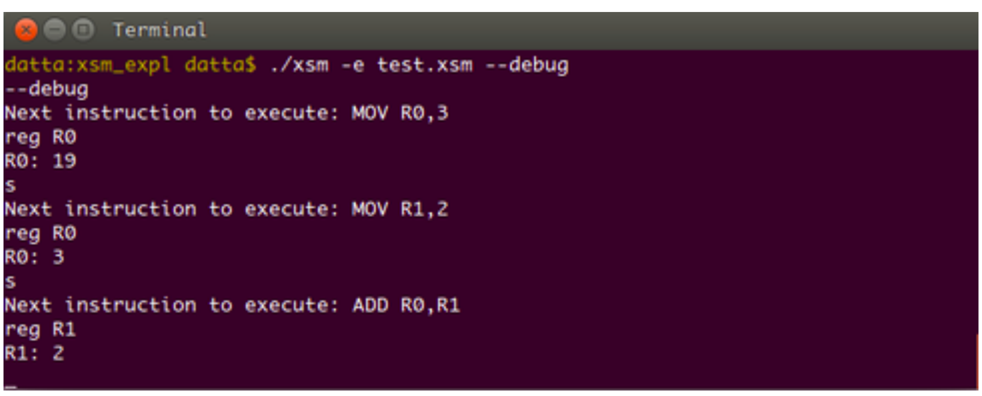
nce we are trying to execute the target_file in debug mode, the simulator ecutes all the instructions present before the BRKP instruction. We will look the status of register R0 before the execution of instruction "MOV R0, 3".



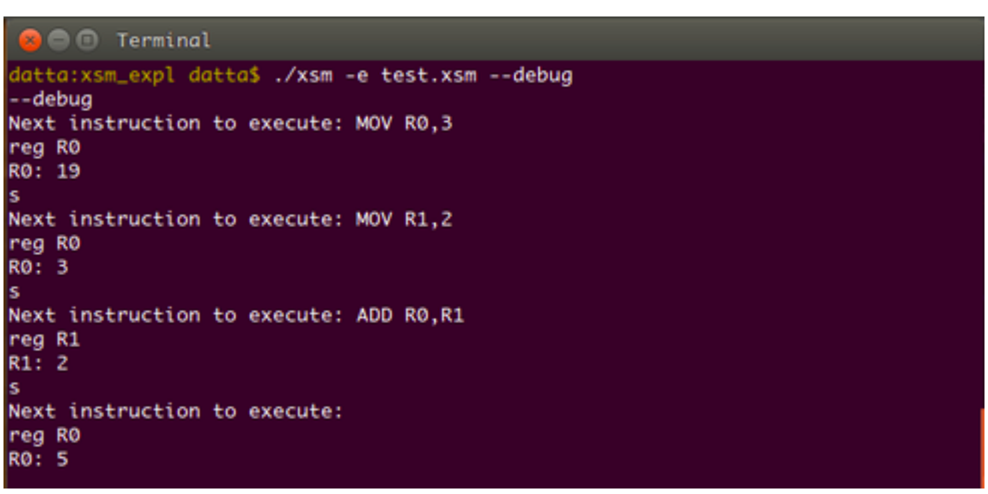
Type "s" and then type "reg R0".



We can see that value 3 is stored in this register. Now try typing "s" and then "reg R1".



Type "s" and then type "reg R0".



We can see that value stored in the register is 5.

Now type reg R0 R1

```
Terminal
datta:xsm_expl datta$ ./xsm -e test.xsm --debug
--debug
Next instruction to execute: MOV R0,3
reg R0
R0: 19
s
Next instruction to execute: MOV R1,2
reg R0
R0: 3
s
Next instruction to execute: ADD R0,R1
reg R1
R1: 2
s
Next instruction to execute:
reg R0
R0: 5
reg R0 R1
R0: 5
R1: 2
```

The command "reg Rn Rm" dislays the contents of all registers from n to m.

Now type reg.

```
Terminal
reg
R0: 5
R1: 2
R2: 40
R3: 53
R4:
R5:
R6:
R7:
R8:
R9:
R10:
R11:
R12:
R13:
R14:
R15:
R16: 35328
R17: 2056
R18:
R19:
P0:
P1:
P2:
P3:
BP:
SP: 4095
IP: 2064
PTBR: 29696
PTLR: 10
EIP:
EC:
EPN:
EMA:
```

ıis command displays the contents of all the machine registers namely IP, ?, BP, PTBR, PTLR, EIP, EC, EPN, EMA, R0-R19 in that order. We will not be ɔncerned with PTBR, PTLR, EIP, EC, EPN and EMA registers which are :cessible only to the OS kernel executing in previliged mode. Note that the ılue of IP register is 2064. IP was initialized to 2056 while loading (entry ɔint value) and after executing four instructions, IP got incremented to)64. The OS kernel had set SP to point 4095 at load time (why?).

ɔte: You would have observed that if you try to continue to execute beyond e last instruction, the simulator flags abnormal program termination. This is ɛcause after executing the last instruction of the program, the simulator crements the instruction pointer and tries to execute the next instruction. ɔwever, since there is no valid instruction stored in the memory, the mulator will enconter an invalid instruction and generate an exception. This sults in transfer of program control to the OS kernel code that will rminate execution and report error. We will see how graceful program rmination is achieved in the next experiment.

xercise 1 : Write an XSM assembly langauge program to find the largest of ree numbers and run it on the simulator. You will learn how to handle the MP instruction while doing this exercise.

xercise 2 : Modify your code generation module to store the result of the evious program in the first location in stack region namely address 4096 nd watch the contents in debug mode after execution.

---

xperiment II : Input/Output using OS system call interface

eading Assignment: Read the low level system call interface of the ABI

this experiment, you will extend the previous stage to print the result of dding two numbers to the console using the low level system call interface ovided by the ABI.

he conceptual point to understand here is that console I/O is handled by the ernel routines of the operating system. Kernel modules execute in privileged ode of execution and can execute special privileged instructions that ccess devices and other resources in a machine.

owever, your XEXE executable program execute in unprivileged mode. Such ograms are called application programs or simply applications. These ograms cannot contain privileged instructions. (If you try to write privileged struction in your program and execute, the machine will raise an exception hen it fetches the instruction and the exception handler module of the OS ernel will terminate the application, flagging an error.) The instructions pecified in the ABI given to you are all unprivileged instructions. (To know ore about privileged instructions in XSM, see ExPOS documentation).

n OS typically provides you with a set of kernel level routines called system alls which your code can invoke for performing console I/O. A system call is voked by a trap instruction (The INT instruction is the trap instruction of the SM machine). Arguments like the system call number specifying the articular OS service (like read/write/program exit) and so on are required. ee details here). The OS specifies how an application must pass arguments nd extract return values from system calls called the calling conventions. enerally, arguments to a system call are passed through the application ogram's stack. These details are written down in the low level system call terface of the ABI.

ote: An OS typically will provide a large number of system calls for various quirements. For our purposes, the relevant system calls are those for onsole read, console write and program exit.

s noted previously, the arguments/return values to/from a system call are ıssed through the application program's stack. Each application maintains a ack region in memory where run-time data can be stored while the ogram executes. The ABI specification stipulates that the stack region of a ogram shall be between memory addresses 4096 and 5119. The application enerally reserves some initial addresses starting from 4096 for storing global iriables in the program (called static allocation) and then initializes the stack the first free memory after those allocated to variables. Arguments to a stem call are pushed into the stack before executing the INT instruction. efore the system call transfer control back (using the IRET instruction), turn values would have been pushed into the stack.

ıplementation :

Experiment I above, you were asked to calculate the sum of two numbers hich are stored in registers. Now, we will see how this number can be inted out into the console.

print the data into the console, we need to:

1. Push the data(using registers) into the stack along with other arguments to the system call.
2. Invoke the write system call using the INT instruction specifying the appropriate interrupt number.

use the stack we need to set the stack pointer(SP). Always SP should point e top value of the stack. The XSM machine increments SP immedietely efore a PUSH operation by XSM machine and hence the first PUSH peration will be storing data to address 4096. So, we need to initialize SP to 095.

```
MOV SP, 4095
```

ow, the arguments to a write operation must be pushed on to the stack: ote that the contents which are to be written to the console are present in e register R0. So, we are not using the register R0. System Call Number : 5 r Write

```
MOV R2, 5
PUSH R2
```

gument 1 : value -2

```
MOV R2, -2
PUSH R2
```

gument 2 : Data to be written (For this example, data is present in R0. So, e are pushing R0)

```
PUSH R0
```

ote : Whenever there is a blank argument or a space to be pushed on to the ack, we follow the convention of pushing R0 on to the stack and the status stack in the figure will be shown using blank argument only.

gument 3 : Blank /* Push any register */

```
PUSH R0
```

orage for Return Value : Push any register

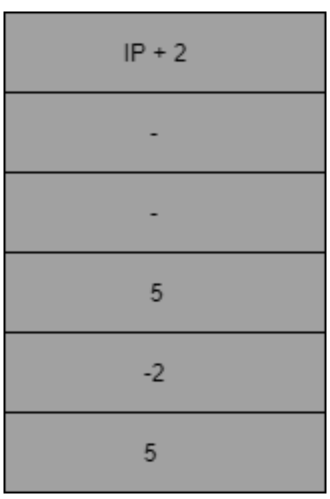
```
PUSH R0
```

ow, invoke the trap instruction to invoke the kernel module for console ıtput. The ABI specifies that the interrupt number must be 7.

terrupt Number : 7 for Write System Call

```
INT 7
```

ıe status of the stack after the INT instruction will look as shown below



ote that in the above diagram argument 2 is 5 because the contents of the gister R0 is 5.

ıe above sequence of instructions will invoke the write system call. Upon ıccessful write operation, the value 0 will be returned to the calling program rough the stack. We assume here that the call will be successful.

ow upon exit from the system call :

1. The return value may be retrievied if required.
2. The stack must be set back to the state before the call as stipulated in the ABI. This is necessary to avoid loss of stack space after each call.

ıe following instructions will do the above.

```
// The following code must be executed after return
from the system call
POP R0 // Pop and save the return value in some
register
POP R1 // Pop and discard the argument3
POP R1 // Pop and discard the argument2
POP R1 // Pop and discard the argument1
POP R1 // Pop out system call number.
```

ne target_file will look as shown below after adding the Write system call.

```
0
2056
0
0
0
0
0
0
MOV R0, 3
MOV R1, 2
ADD R0, R1
MOV SP, 4095
MOV R2, 5
PUSH R2
MOV R2, -2
PUSH R2
PUSH R0
PUSH R2
PUSH R2
INT 7
POP R0
POP R1
POP R1
POP R1
POP R1
```

target_file after adding Write system call hosted with ❤ by **GitHub** **view raw**

ow, having generated the executable program run the program using the mulator. To run the program, follow the instructions given below.

1. open terminal and navigate to the XSM simulator folder.
2. Type "./xsm -e < relative path to target_file.xsm> "

ou can see that the value 5 is printed on the console.

bserve that the simulator flagged an error after the last statement was ecuted. This happened because after executing the last valid instruction in e program, the simulator had no idea that the program had ended and ence tried to fetch the next instruction from memory. However, since there no valid insturction in that memory location, the machine raises an xception [see here for exceptions of XSM] and control was transfered to an xception handler routine of the OS kernel. Typically, in a multitasking nvironment, the OS will terminate the program, reclaim resources allocated it and schedule some other process.

he exception handler routine of the XSM simulator given to you is designed print an error message and terminate the simulation.

he "proper" way to exit the application is to invoke the exit system call. This ill inform the OS that the program has finished execution. The exit system all code of a typical OS kernel will "gracefully" exit the application and chedule other programs for execution. The OS will never return to the pplication that invokes the exit system call.

he exit system call routine of the XSM simulator given to you will print a essage indicating successful program execution and terminate the mulation.

nportant Note : The system call code for read, write and exit will modify ome of the registers R0, R1 etc. Hence, if you had stored some value into ese registers before the call, those values will be lost during the call. Hence, you wish to restore the values of the registers, the following procedure ust be adopted.

1. Push all registers in current use into the stack. For example, if you wish to retain values of R0, R1 and R2 after the call, push these three registers into the stack.
2. Now, follow procedure outlined earlier to make the system call.
3. After return POP out the registers saved in the stack. (Note that you must pop the registers out in reverse order of push).

xercise 3 : Follow the instructions in the low level system call interface of e ABI to invoke the exit system call after the console output in your revious program.

o far, there was no need to allocate memory for storing variables as all the ata involved were stored in registers. The next exercise requires allocation of orage for variables. Suppose you want to read a number from the console, en the address of a memory location must be passed as the second gument to the read system call (INT 6). The system call will place the input ata into the memory address received as the second argument.

uppose you need three variables to be read, then you may reserve the first ree words of the stack (memory addresses 4096, 4097 and 4098) for those riables. As some portion of the stack is now reserved for variables, the initial lue of the stack pointer may be set such that the run time stack begins oove the reserved region. (SP may be set to 4098 here). This ensures that e reserved region is not over-written when the program pushes data into e stack. To read a variable from input, you need to push the address of the emory location reserved for the variable as the second argument (4096 for e first, 4097 for the second, 4098 for the third.) and invoke the read system ıll. With this strategy, the following exercise can be easily solved.

‹ercise 4 : Write an XSM assembly language program to read three numbers ɔm the console and print the largest. (Invoke the read system call for ɔnsole input.) Caution : Ensure that you understand the important note ritten above before starting with the implementation.

‹ercise 5 : Write an XSM assembly language program to read numbers until a ːro is entered and print their sum.

ıportant Note : Exercises 4 and 5 essentially ask you to handle conditional ıd iterative constructs in assembly language. These exercises give insight to how machine code must be genrated for if-then-else and while-do ɔnstructs of programming languages.

‹periment III : Understanding the Library Interface

'erequisite Reading : Read and understand the library interface.

this experiment, you will learn how to implement the library interface ipulated in the ABI for supporting read, write and exit system calls.

ıe memory address space model of a program reserves the first 1024 words ' the address space of a program to load a library. Here we explain the ırpose of the library.

we consider C programs, almost every program uses the routines in the ɔrary *stdio.h*. Since in a computer system, several application programs will ː running concurrently, it is a good idea to have the code for *stdio.h* loaded ıce at bootstrap time into some region of the physical memory and link this emory to the address space of each program's standard library region at ad time. This code will be designed once and shared between all ɔplications.

ıe ABI specifies that the ExpOS library for the XSM machine must be linked address 0 to 1023. The XSM simulator given to you will load the contents ' the file *library.lib* to the addresses 0 - 1023 of your program. The ABI ipulates that the library must support functions for read, write and exit. (The ɔrary also must contain functions Alloc, Free and Initialize which will not be scussed here.)

ɔ access any library function, an application must transfer control to the ɔde at memory address 0 using the instruction CALL. This is the first emory address in the library region. The arguments to the call specify which ɔrary function is being invoked. The library interface is specified here.

ı application program can execute read, write and exit functions through e library. This means that once the library is implemented, application ograms can call the library (CALL 0) to perform read, write and exit ɔerations by passing appropriate function code and arguments. Internally, e library contains code that traps to the Os kernel.

ne might naturally raise the question – why should we route the system ılls through the library than call them directly as was done so far. There are everal advantages in using the library. At a later time, if the interrupt number r an OS service – say write – gets modified, only the library needs to be placed. The compiler need not be modified, nor application programs need -compilation. Thus, the library provides an abstraction that hides low level etails from the compiler and the application.

nplementation :

this experiment, we implement the program of Experiment II using the ›rary.

ecall that to print the data into the console, we need to:

1. Push the data(using registers) into the stack along with other arguments to the system call.
2. Invoke the write system call using the INT instruction specifying the appropriate interrupt number.

ne step 1 will remain same as in the above experiment. We will implement ep 2 using library interface.

› use the stack we need to set the stack pointer(SP). Always SP should point e top value of the stack. The XSM machine increments SP immedietely efore a PUSH operation by XSM machine and hence the first PUSH peration will be storing data to address 4096. So, we need to initialize SP to)95.

```
MOV SP, 4095
```

ow, the arguments to a write operation must be pushed on to the stack: ote that the contents which are to be written to the console are present in e register R0. So, we are not using the register R0. Function Code : "Write"

```
MOV R2, "Write"
PUSH R2
```

gument 1 : value -2 (see ABI for specification)

```
MOV R2, -2
PUSH R2
```

gument 2 : Data to be written (For this example, data is present in R0. So, e are pushing R0)

```
PUSH R0
```

gument 3 : Blank /* Push any register */

```
PUSH R0
```

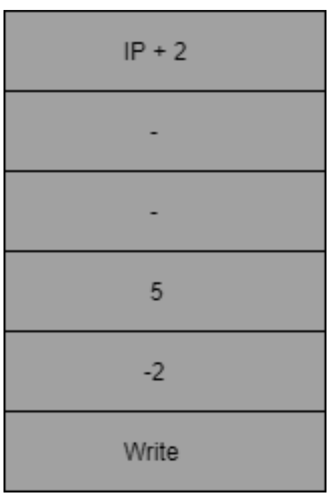
orage for Return Value : Push any register

```
PUSH R0
```

ne main difference between the system call interface and the library terface come at the next instruction. If we were using the system call terface, we would have called the interrupt for write (INT 7). Here instead, e will always use CALL 0 to transfer control to the library. (Remember that orary file is loaded in the memory addresses 0 - 1023.). The library nderstands that the action requested is a write operation by looking at the nction code passed as argument (in this case "Write"). If the action quested is a read operation, then "Read" is passed, and so on.

```
CALL 0
```

ne status of the stack after the INT instruction will look as shown below



ote that in the above diagram argument 2 is 5 because the contents of the gister R0 is 5.

ne above sequence of instructions will invoke the write system call from the orary. Upon successful write operation, the library must retuen value 0 to the alling program through the stack. We assume here that the call will be uccessful.

ow upon exit from the call :

1. The return value may be retrieved if required.
2. The stack must be set back to the state before the call as stipulated in the ABI. This is necessary to avoid loss of stack space after each call.

ne following instructions will do the above.

```
// The following code must be executed after return
from the system call
```

```
POP R0 // Pop and save the return value in some
register
POP R1 // Pop and discard the argument3
POP R1 // Pop and discard the argument2
POP R1 // Pop and discard the argument1
POP R1 // Pop out the function code.
```

he target_file will look as shown below after adding the Write system call.

```
0
2056
0
0
0
0
0
0
MOV R0, 3
MOV R1, 2
ADD R0, R1
MOV SP, 4095
MOV R1, "Write"
PUSH R1
MOV R1, -2
PUSH R1
PUSH R0
PUSH R1
PUSH R1
CALL 0
POP R0
POP R1
POP R1
POP R1
POP R1
```

target_file after adding Write system call through library interface hosted with ❤ **view raw**
oy **GitHub**

ow we need to write logic in the library file to handle the request for Write stem call. Earlier we have created the "library.lib" file with RET instruction one. Now we will edit this file.

hen CALL 0 is executed, the IP+2 is pushed on the top of the stack and IP is t to 0. Therefore now IP points to the first instruction of the library file.

nce every call to the library points IP to 0 address, we need to have logic to stinguish the type of request at this address. The information regarding the pe of request can be found from the function code that is pushed as an gument on the stack. The function code can be obtained from the stack sing SP - 5[Why?].

this experiment we are dealing only with "Write" system call, so we will ave our discussion only specific to it. After recognising the type of request e need to handle the request in the same way as we did in the previous ‹periment using system call interface. The arguments to be passed in the 'stem call interface are system call number(5),-2, register holding the value › be written (It is at SP - 3) and a blank argument. The library simply has to ›py the arguments 1 to 3 to the system call without any modification.

et the function code from the stack and compare if it is equal to "Write" and ›t the arguments from the stack.

```
MOV R1, SP
MOV R2, 5
SUB R1, R2
MOV R2, "Write"
MOV R1, [R1]
EQ R1, R2
JZ R1, 62
MOV R1, SP
MOV R2, 4
SUB R1, R2
MOV R2, [R1] //argument 1 at SP - 4
ADD R1, 1
MOV R3, [R1] //argument 2 at SP - 3
ADD R1, 1
MOV R4, [R1] //argument 3 at SP - 2
```

nce you get the arguments, the procedure to invoke the system call is same › we did in the earlier experiment.

```
/* pushing arguments and space for return value for
system call */
MOV R5, 5 //System call number 5 for Write
PUSH R5
PUSH R2 //R2 contains [SP - 4] = -2
PUSH R3 //R3 contains [SP - 3] = 5 // Contains Value
PUSH R4 //R4 contains [SP - 2] = blank
PUSH R5 // space for return value

/* trap instruction */
INT 7

/* Pop the arguments and return value */
POP R1 //Pop and save return value
POP R2 //Pop and discard
POP R2 //Pop and discard
POP R2 //Pop and discard
POP R2 //Pop and discard
```

ıe return value of the library is the same as the return value of the system ıll. Hence the library must copy the return value of the system call back to e application as its own return value through the stack.

```
/* Storing the return value on the top of the stack
*/
MOV R2, SP
MOV R3, 1
SUB R2, R3
MOV [R2], R1
// [SP - 1] = R1 (Return value was popped to R1
earlier) RET
```

ıe library.lib file will look as shown below after adding the code to handle rite system call.

```
MOV R1, SP
MOV R2, 5
SUB R1, R2
MOV R2, "Write"
MOV R1, [R1]
EQ R1, R2
JZ R1, 62
MOV R1, SP
MOV R2, 4
SUB R1, R2
MOV R2, [R1]
ADD R1, 1
MOV R3, [R1]
ADD R1, 1
MOV R4, [R1]
MOV R5, 5
PUSH R5
PUSH R2
PUSH R3
PUSH R4
PUSH R5
INT 7
POP R1
POP R2
POP R2
POP R2
POP R2
MOV R2, SP
MOV R3, 1
SUB R2, R3
MOV [R2], R1
RET
```

library.lib with Write System Call hosted with ❤ by **GitHub** **view raw**

nportant Note : Your library code for read, write and exit may modify some f the registers R0, R1 etc. Hence, if you had stored some values into these gisters before the call, those values will be lost while the library code is xecuted. To safely restore the values of the registers, the following ocedure must be adopted.

1. Push all registers in current use into the stack. For example, if you wish to retain values of R0, R1 and R2 after the call, push these three registers into the stack.
2. Now, code the library call and return steps.
3. After return POP out the registers saved in the stack. (Note that you must pop the registers out in reverse order of push).

xercise 6 : Implement read() and exit() functions of the library.

xercise 7 : Modify the program to read three numbers from the console and int the largest to perform I/O using the library interface.

xercise 8 : Modify the program to read numbers until a zero is entered from e console and print their sum to perform I/O using the library interface.

Contributed By : K Murali Krishnan