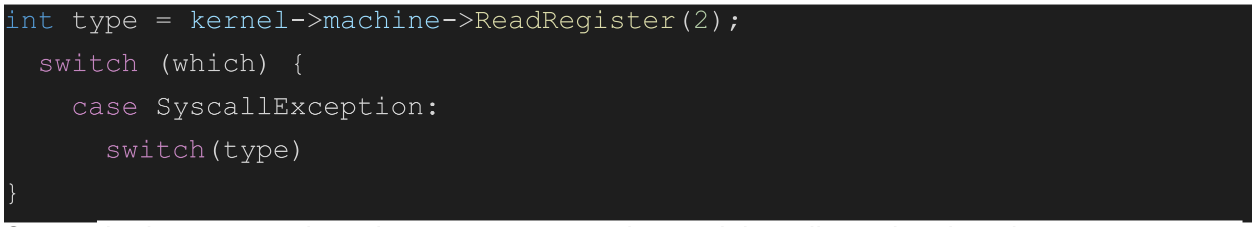
Q1

Step 6.
Because ExceptionHandler is called by a user program, the entry point into the Nachos kernel. And step 6 the library procedure executes a TRAP instruction to switch from user mode to kernel mode and start execution at a fixed address within the kernel.

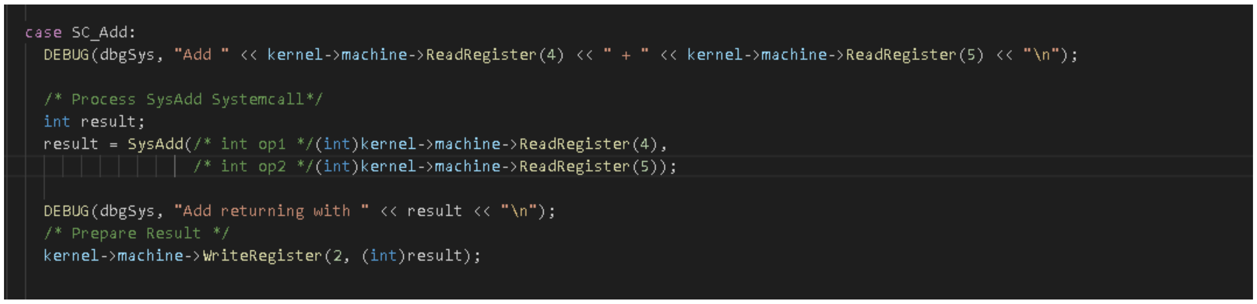
Q2

Step 6,7,8,9.10,11
Step 6: will deliver ExceptionHandler()



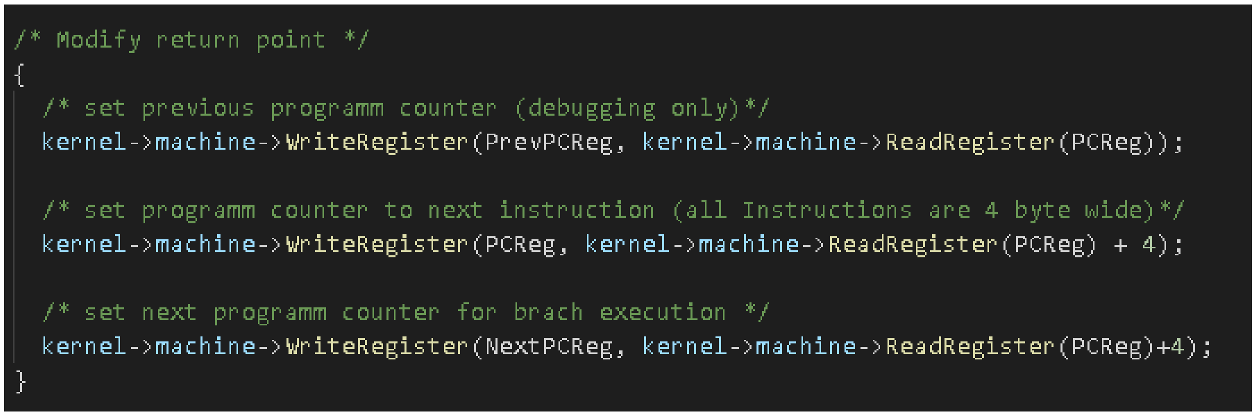
```
int type = kernel->machine->ReadRegister(2);
  switch (which) {
    case SyscallException:
      switch(type)
}
```

Step 7:the kernel examines the system call number and then dispatches it to the correct system call handler. This correct number is given in the table of system call handlers by pointers referenced at the system call number.
For example: SC_add is an situation.



```
case SC_Add:
  DEBUG(dbgSys, "Add " << kernel->machine->ReadRegister(4) << " + " << kernel->machine->ReadRegister(5) << "\n");

  /* Process SysAdd Systemcall*/
  int result;
  result = SysAdd(/* int op1 */(int)kernel->machine->ReadRegister(4),
                  /* int op2 */(int)kernel->machine->ReadRegister(5));

  DEBUG(dbgSys, "Add returning with " << result << "\n");
  /* Prepare Result */
  kernel->machine->WriteRegister(2, (int)result);
```

Step 8: the system call handler runs, in this case is SysAdd() function.
Step 9: the operation is completed, and the user is given back control once the TRAP instruction is set, which is defined kernel->machine->WriteRegister(2,(int)result; in this case.
Step 10 and 11:
To return to user program and restore the call stage, which is implemented by following code:



```
/* Modify return point */
{
  /* set previous programm counter (debugging only)*/
  kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));

  /* set programm counter to next instruction (all Instructions are 4 byte wide)*/
  kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) + 4);

  /* set next programm counter for brach execution */
  kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCReg)+4);
}
```

Q3

The function is kernel->machine->ReadRegister(), which can call the user program to put the parameter into stack and kernel can reach it from stack.

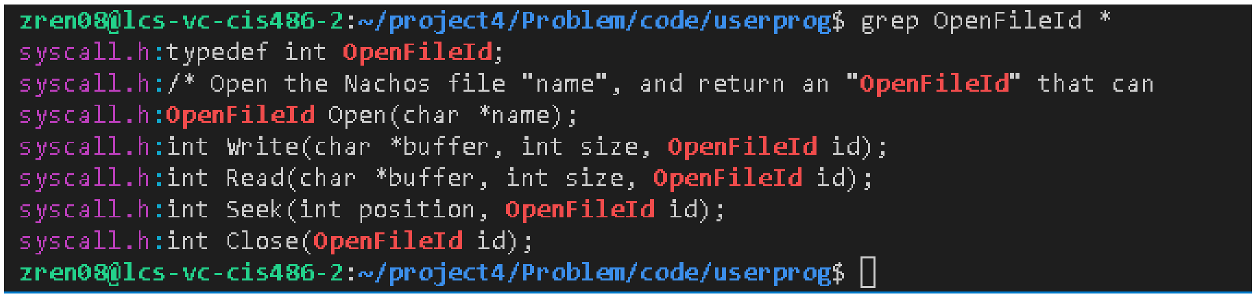
```
case SC_Add:
          DEBUG(dbgSys, "Add " << kernel->machine->ReadRegister(4) << " +
" << kernel->machine->ReadRegister(5) << "\n");

          /* Process SysAdd Systemcall*/
          int result;
          result = SysAdd(/* int op1
*/(int)kernel->machine->ReadRegister(4),
                          /* int op2
*/(int)kernel->machine->ReadRegister(5));

          DEBUG(dbgSys, "Add returning with " << result << "\n");
          /* Prepare Result */
          kernel->machine->WriteRegister(2, (int)result);


          break;
```

Q4

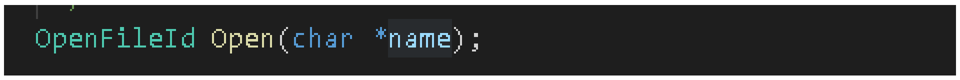
Syscall.h defines OpenFileId at line 100 and ConsoleOutput at line 109.OpenFileId is just a file descriptor and its type is defined in syscall.h at line 100. ConsoleOutput is a const parameter defined in line 109 of syscall.h

OpenFileId: File system operations: Create, Remove, Open, Read, Write, Close. These functions are patterned after UNIX -- files represent. Both files *and* hardware I/O devices. However, the Nachos file system has a stub implementation, which can be used to support these system calls if the regular Nachos file system has not been implemented.

OpenFileId is a unique identifier for an open Nachos file.

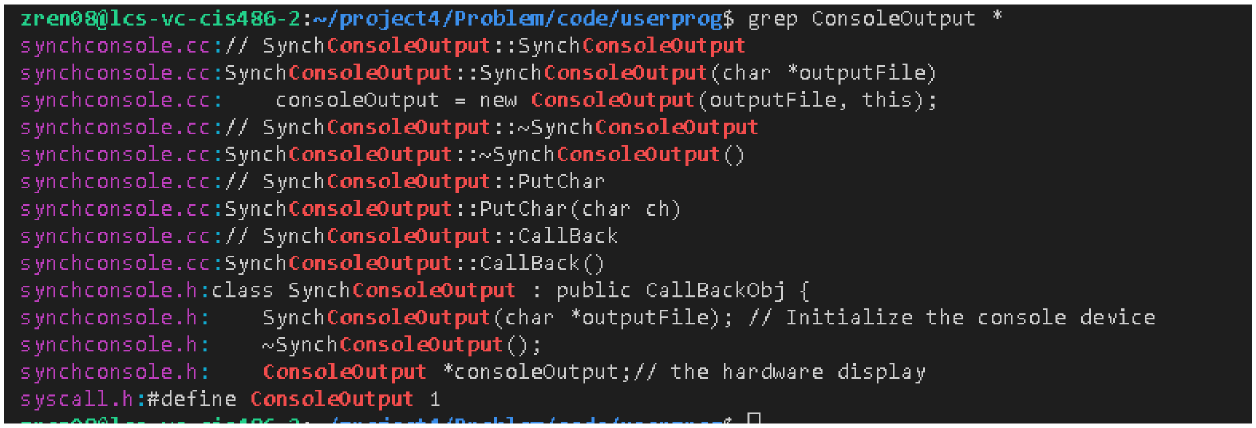


```
zren08@lcs-vc-cis486-2:~/project4/Problem/code/userprog$ grep OpenFileId *
syscall.h:typedef int OpenFileId;
syscall.h:/* Open the Nachos file "name", and return an "OpenFileId" that can
syscall.h:OpenFileId Open(char *name);
syscall.h:int Write(char *buffer, int size, OpenFileId id);
syscall.h:int Read(char *buffer, int size, OpenFileId id);
syscall.h:int Seek(int position, OpenFileId id);
syscall.h:int Close(OpenFileId id);
zren08@lcs-vc-cis486-2:~/project4/Problem/code/userprog$
```

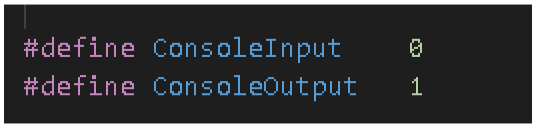


```
OpenFileId Open(char *name);
```

ConsoleOutput:: When an address space starts up, it has two open files, representing keyboard input and display output (in Unix terms, stdin and stout). Read and Write can be used directly on these, without first opening the console device.
And ConsoleOutput is defined as 1.

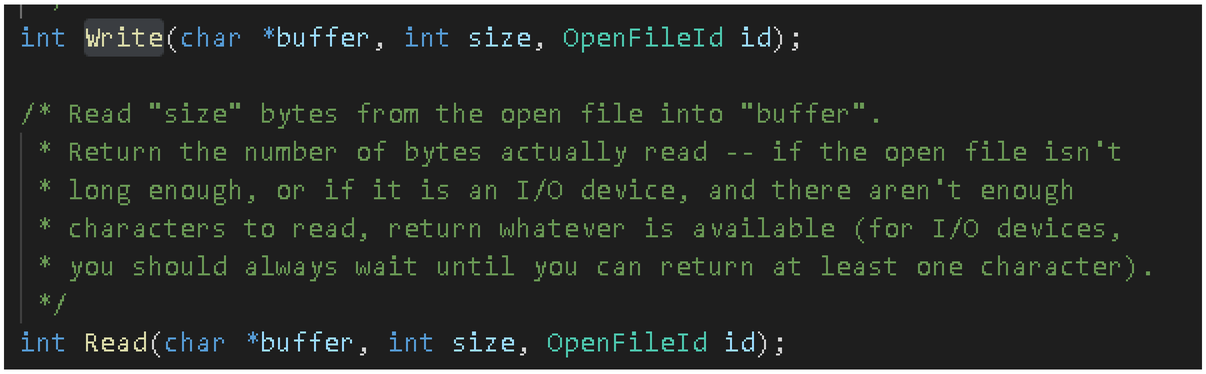


```
zren08@lcs-vc-cis486-2:~/project4/Problem/code/userprog$ grep ConsoleOutput *
synchconsole.cc:// SynchConsoleOutput::SynchConsoleOutput
synchconsole.cc:SynchConsoleOutput::SynchConsoleOutput(char *outputFile)
synchconsole.cc:    consoleOutput = new ConsoleOutput(outputFile, this);
synchconsole.cc:// SynchConsoleOutput::~SynchConsoleOutput
synchconsole.cc:SynchConsoleOutput::~SynchConsoleOutput()
synchconsole.cc:// SynchConsoleOutput::PutChar
synchconsole.cc:SynchConsoleOutput::PutChar(char ch)
synchconsole.cc:// SynchConsoleOutput::CallBack
synchconsole.cc:SynchConsoleOutput::CallBack()
synchconsole.h:class SynchConsoleOutput : public CallBackObj {
synchconsole.h:    SynchConsoleOutput(char *outputFile); // Initialize the console device
synchconsole.h:    ~SynchConsoleOutput();
synchconsole.h:    ConsoleOutput *consoleOutput;// the hardware display
syscall.h:#define ConsoleOutput 1
```



```
#define ConsoleInput    0
#define ConsoleOutput   1
```

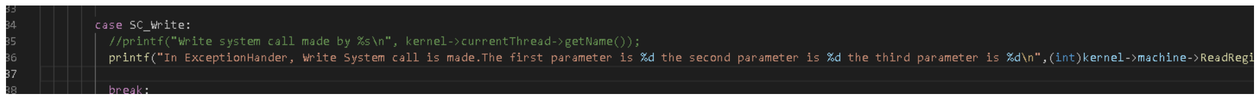
Q5
The write is defined in userprog/syscall.h
Void Write(char *buffer, int size, OpenFileId id)
Write “size” bytes from “buffer” to the open file.
Buffer is the buffer which will be written to the target file.
Size is the number of characters need to write.
OpenFileId is the file descriptor of the target file
Writing the size of 18 chars from str into the output, which is defined as ConsoleOutput.



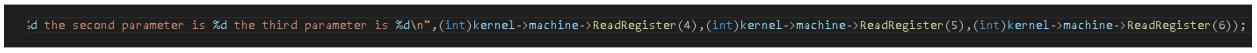
```
int Write(char *buffer, int size, OpenFileId id);

/* Read "size" bytes from the open file into "buffer".
 * Return the number of bytes actually read -- if the open file isn't
 * long enough, or if it is an I/O device, and there aren't enough
 * characters to read, return whatever is available (for I/O devices,
 * you should always wait until you can return at least one character).
 */
int Read(char *buffer, int size, OpenFileId id);
```

Q6
Due to x,y and z are the values of the parameter in integers. We define the output should be (int). We change the code in SC-Write to get its’ string(the str”Hello from prog1”) from first register (register(4)) and then from register (register(register(5)) to get its buffer size and finally get the OpenFileId from register(register(6)).

```
case SC_Write:
          //printf("Write system call made by %s\n", 
kernel->currentThread->getName());
          printf("In ExceptionHander, Write System call is made.The first 
parameter is %d the second parameter is %d the third parameter is 
%d\n",(int)kernel->machine->ReadRegister(4),(int)kernel->machine->ReadRegi
ster(5),(int)kernel->machine->ReadRegister(6));

          break;
```

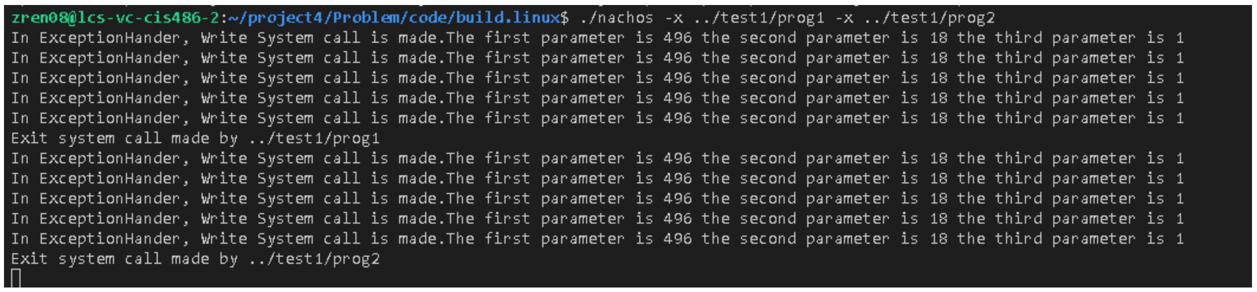


```
case SC_Write:
  //printf("Write system call made by %s\n", kernel->currentThread->getName());
  printf("In ExceptionHander, Write System call is made.The first parameter is %d the second parameter is %d the third parameter is %d\n",(int)kernel->machine->ReadRegi

  break;
```



```
%d the second parameter is %d the third parameter is %d\n",(int)kernel->machine->ReadRegister(4),(int)kernel->machine->ReadRegister(5),(int)kernel->machine->ReadRegister(6));
```

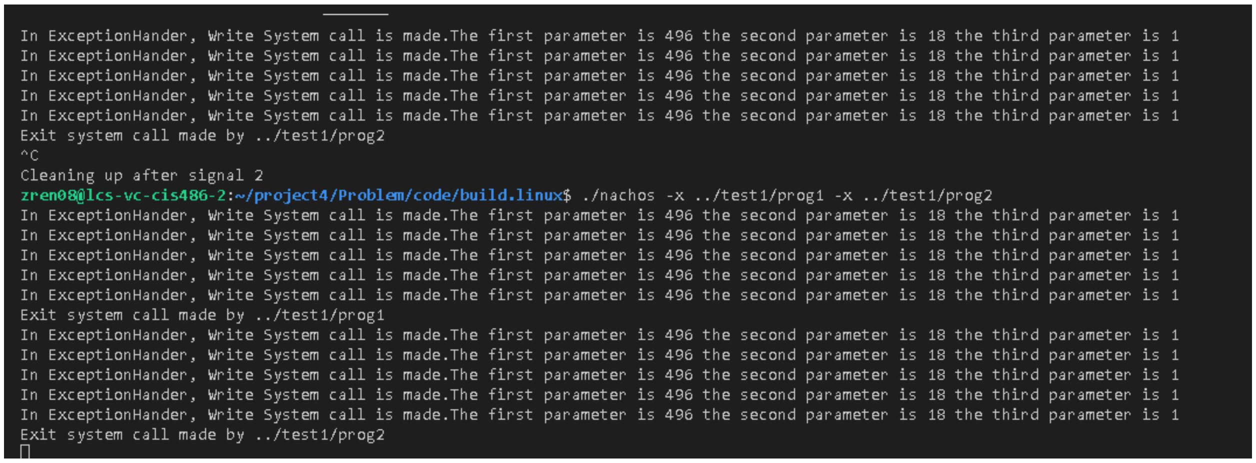
The output is like:



```
zren08@lcs-vc-cis486-2:~/project4/Problem/code/build.linux$ ./nachos -x ../test1/prog1 -x ../test1/prog2
In ExceptionHander, Write System call is made.The first parameter is 496 the second parameter is 18 the third parameter is 1
In ExceptionHander, Write System call is made.The first parameter is 496 the second parameter is 18 the third parameter is 1
In ExceptionHander, Write System call is made.The first parameter is 496 the second parameter is 18 the third parameter is 1
In ExceptionHander, Write System call is made.The first parameter is 496 the second parameter is 18 the third parameter is 1
In ExceptionHander, Write System call is made.The first parameter is 496 the second parameter is 18 the third parameter is 1
Exit system call made by ../test1/prog1
In ExceptionHander, Write System call is made.The first parameter is 496 the second parameter is 18 the third parameter is 1
In ExceptionHander, Write System call is made.The first parameter is 496 the second parameter is 18 the third parameter is 1
In ExceptionHander, Write System call is made.The first parameter is 496 the second parameter is 18 the third parameter is 1
In ExceptionHander, Write System call is made.The first parameter is 496 the second parameter is 18 the third parameter is 1
In ExceptionHander, Write System call is made.The first parameter is 496 the second parameter is 18 the third parameter is 1
Exit system call made by ../test1/prog2
```

The first parameter 496 means the buf address in Write(str,18,output), the second parameter 18 means the size of the str buffer. And the third parameter means the OpenFileId, 1 represents getting the file(Opening the file)

Q7


```
In ExceptionHander, Write System call is made.The first parameter is 496 the second parameter is 18 the third parameter is 1
In ExceptionHander, Write System call is made.The first parameter is 496 the second parameter is 18 the third parameter is 1
In ExceptionHander, Write System call is made.The first parameter is 496 the second parameter is 18 the third parameter is 1
In ExceptionHander, Write System call is made.The first parameter is 496 the second parameter is 18 the third parameter is 1
In ExceptionHander, Write System call is made.The first parameter is 496 the second parameter is 18 the third parameter is 1
Exit system call made by ../test1/prog2
^C
Cleaning up after signal 2
zren08@lcs-vc-cis486-2:~/project4/Problem/code/build.linux$ ./nachos -x ../test1/prog1 -x ../test1/prog2
In ExceptionHander, Write System call is made.The first parameter is 496 the second parameter is 18 the third parameter is 1
In ExceptionHander, Write System call is made.The first parameter is 496 the second parameter is 18 the third parameter is 1
In ExceptionHander, Write System call is made.The first parameter is 496 the second parameter is 18 the third parameter is 1
In ExceptionHander, Write System call is made.The first parameter is 496 the second parameter is 18 the third parameter is 1
In ExceptionHander, Write System call is made.The first parameter is 496 the second parameter is 18 the third parameter is 1
Exit system call made by ../test1/prog1
In ExceptionHander, Write System call is made.The first parameter is 496 the second parameter is 18 the third parameter is 1
In ExceptionHander, Write System call is made.The first parameter is 496 the second parameter is 18 the third parameter is 1
In ExceptionHander, Write System call is made.The first parameter is 496 the second parameter is 18 the third parameter is 1
In ExceptionHander, Write System call is made.The first parameter is 496 the second parameter is 18 the third parameter is 1
In ExceptionHander, Write System call is made.The first parameter is 496 the second parameter is 18 the third parameter is 1
Exit system call made by ../test1/prog2
```

The first parameter value is 496.
And they are the same value but identical for three parameters, because they are in different programs.
They are the virtual addresses defined in each program.
The first parameter 496 means the str's address, the second parameter 18 means the size of the str buffer. And the third parameter means the OpenFileId, 1 represents getting the file. Because OpenFileId is a bool

Q8:
translate.cc mainly do the job that routines to translate virtual addresses to physical addresses. Software sets up a table of legal translations. We look up in the table on every memory reference to find the true physical memory location.

Machine::ReadMem mainly read "size" (1, 2, or 4) bytes of virtual memory at "addr" into the location pointed to by "value". Will return FALSE if the translation step from virtual to physical memory failed.
The three parameters value has the following function:
ReadMem(int addr, int size, int *value)
"addr" -- the virtual address to read from
"size" -- the number of bytes to read (1, 2, or 4)
"value" -- the place to write the result (It is a pointer)

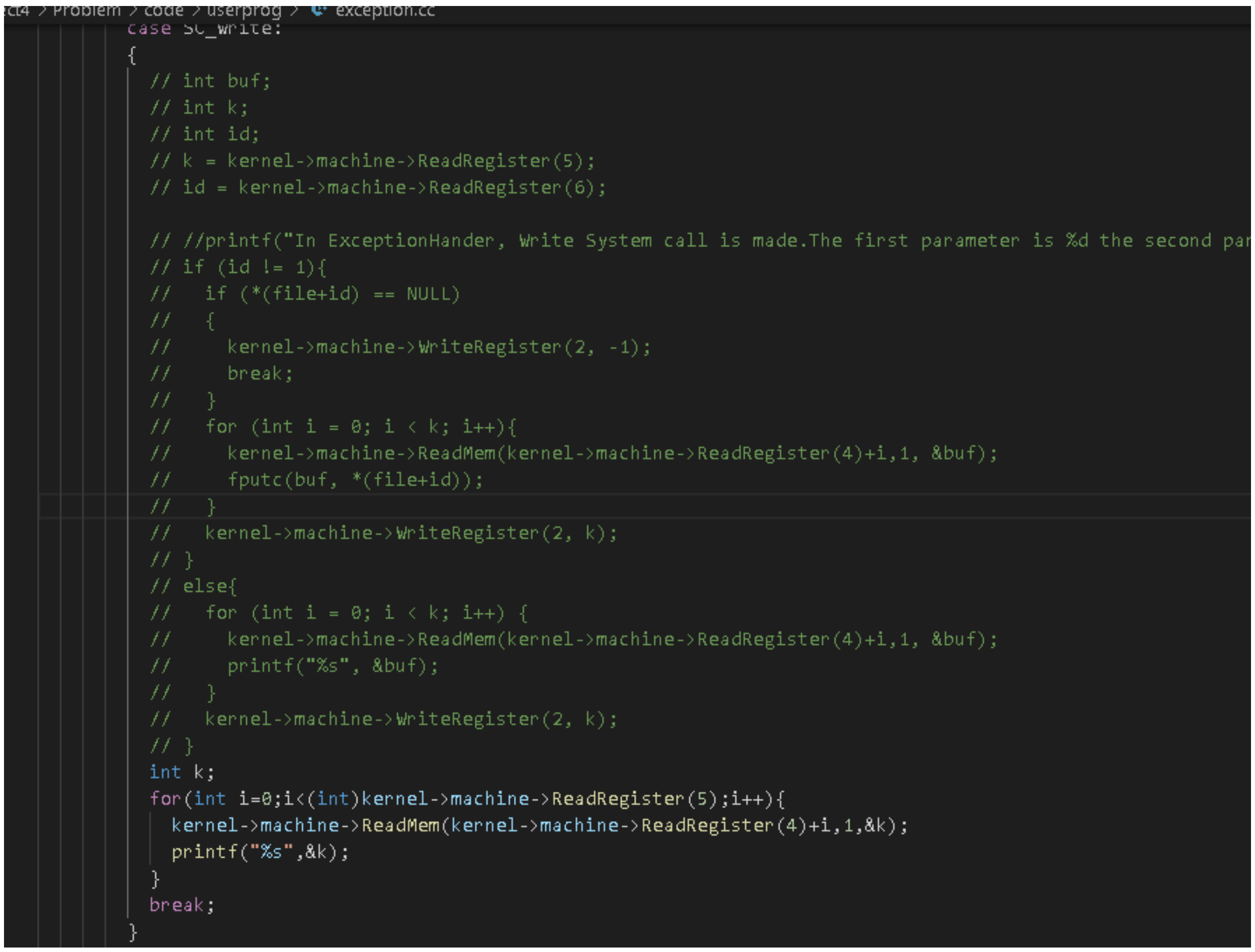
Q9:
This line will set next program counter for brach execution.
Adding 4 is because MIPS instructions are usually 32-bit wide, memories are byte addressable. Therefore, instructions take four 8-bit address spaces in the memory. So PC=PC+4 to get the next program counter.

Implementations

SC_write code:
The Register5 stores the buf size, which is k in my code.
We can know that Register6 stores whether we get or not the OpenfileId. So based the id, we can do the different function.
If both file and Openfile are 0 write nothing to the Writereigister, then break.
Else we read the str for the size of 1 each time, and put it into the buf, loops it for k times.
After finishing the loop, put the k into writeRegister2.

Else output the whole buf to console to show the result.

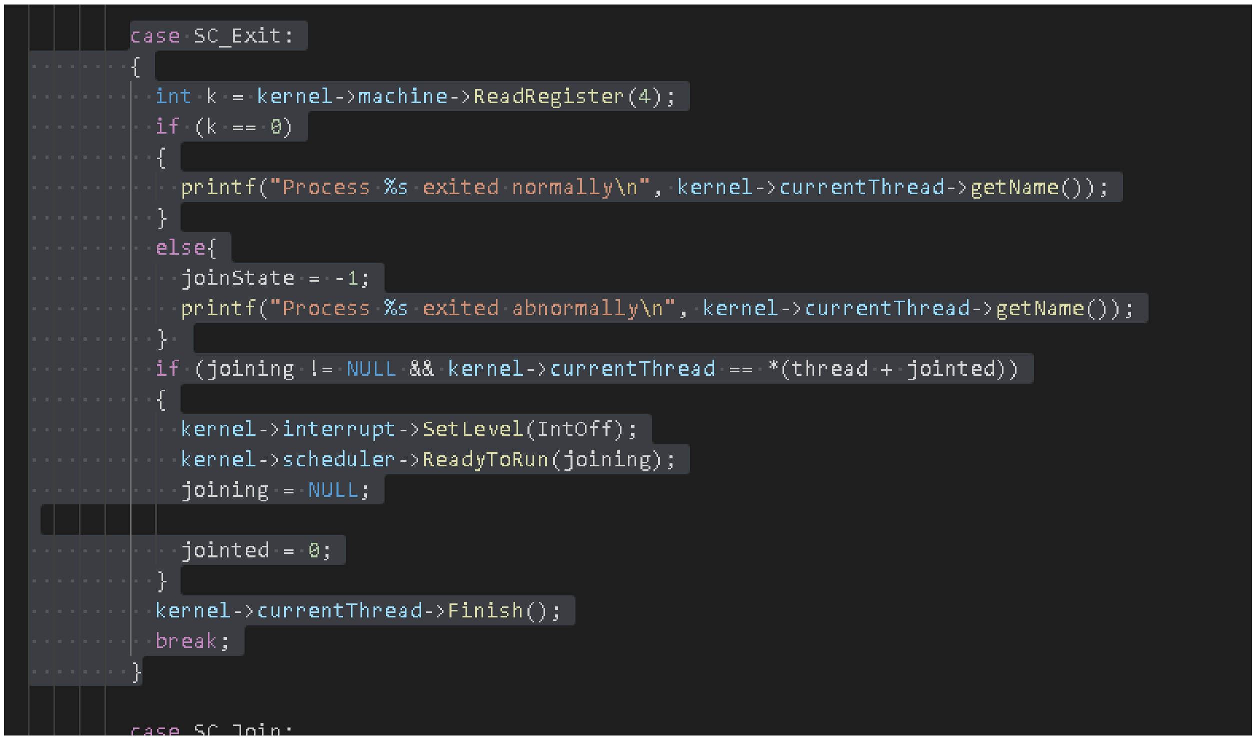
```
        case SC_Write:
          int k;
          for(int i=0;i<(int)kernel->machine->ReadRegister(5);i++){

kernel->machine->ReadMem(kernel->machine->ReadRegister(4)+i,1,&k);
            printf("%s",&k);
          }
          break;
```



```
case SC_Write:
{
  // int buf;
  // int k;
  // int id;
  // k = kernel->machine->ReadRegister(5);
  // id = kernel->machine->ReadRegister(6);

  // //printf("In ExceptionHander, Write System call is made.The first parameter is %d the second par
  // if (id != 1){
  //   if (*(file+id) == NULL)
  //   {
  //     kernel->machine->WriteRegister(2, -1);
  //     break;
  //   }
  //   for (int i = 0; i < k; i++){
  //     kernel->machine->ReadMem(kernel->machine->ReadRegister(4)+i,1, &buf);
  //     fputc(buf, *(file+id));
  //   }
  //   kernel->machine->WriteRegister(2, k);
  // }
  // else{
  //   for (int i = 0; i < k; i++) {
  //     kernel->machine->ReadMem(kernel->machine->ReadRegister(4)+i,1, &buf);
  //     printf("%s", &buf);
  //   }
  //   kernel->machine->WriteRegister(2, k);
  // }
  int k;
  for(int i=0;i<(int)kernel->machine->ReadRegister(5);i++){
    kernel->machine->ReadMem(kernel->machine->ReadRegister(4)+i,1,&k);
    printf("%s",&k);
  }
  break;
}
```

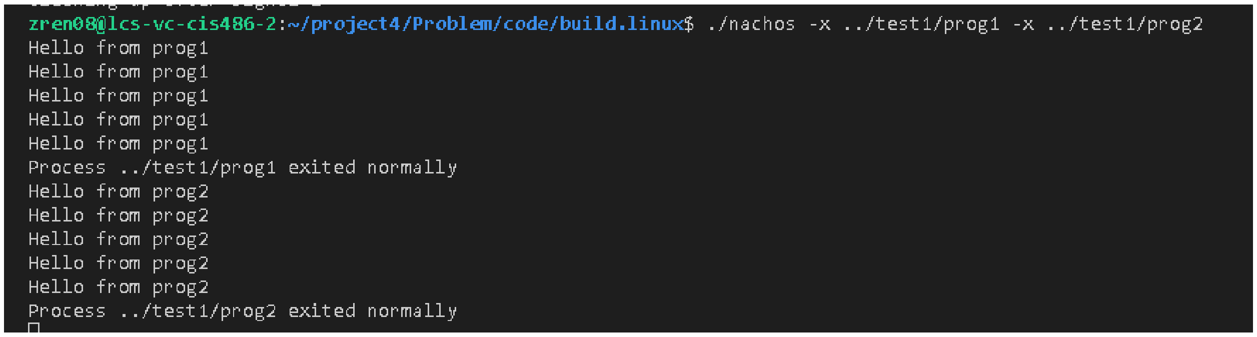
SC_Exit function code:
We directly get the status of the process, then based on normally exited or not to output their result with their name.

```
case SC_Exit:
        {
          int k = kernel->machine->ReadRegister(4);
          if (k == 0)
          {
            printf("Process %s exited normally\n",
kernel->currentThread->getName());
          }
          else{
            joinState = -1;
            printf("Process %s exited abnormally\n",
kernel->currentThread->getName());
          }
          if (joining != NULL && kernel->currentThread == *(thread +
jointed))
          {
            kernel->interrupt->SetLevel(IntOff);
            kernel->scheduler->ReadyToRun(joining);
            joining = NULL;

            jointed = 0;
          }
          kernel->currentThread->Finish();
          break;
        }
```



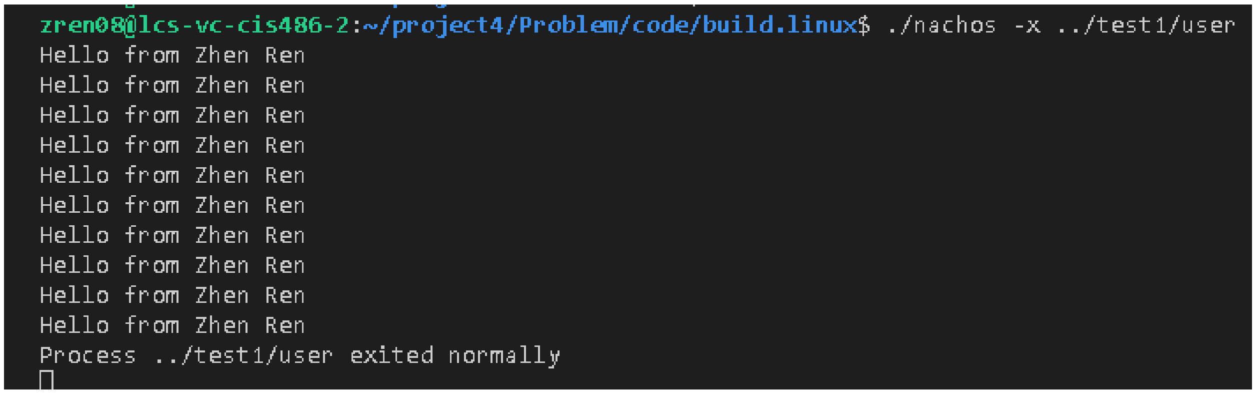
```
        case SC_Exit:
        {
          int k = kernel->machine->ReadRegister(4);
          if (k == 0)
          {
            printf("Process %s exited normally\n", kernel->currentThread->getName());
          }
          else{
            joinState = -1;
            printf("Process %s exited abnormally\n", kernel->currentThread->getName());
          }
          if (joining != NULL && kernel->currentThread == *(thread + jointed))
          {
            kernel->interrupt->SetLevel(IntOff);
            kernel->scheduler->ReadyToRun(joining);
            joining = NULL;

            jointed = 0;
          }
          kernel->currentThread->Finish();
          break;
        }

        case SC_Join:
```

Q10
Prog1 and prog2’s result



```
zren08@lcs-vc-cis486-2:~/project4/Problem/code/build.linux$ ./nachos -x ../test1/prog1 -x ../test1/prog2
Hello from prog1
Hello from prog1
Hello from prog1
Hello from prog1
Hello from prog1
Process ../test1/prog1 exited normally
Hello from prog2
Hello from prog2
Hello from prog2
Hello from prog2
Hello from prog2
Process ../test1/prog2 exited normally
```

Firstly run prog1 in SC_write put the hello from prog1 into buf five time then output the whole buf in str format. Then call the SC_exit to output the process status, exited normally.
And do the same work for prog2.



```
zren08@lcs-vc-cis486-2:~/project4/Problem/code/build.linux$ ./nachos -x ../test1/user
Hello from Zhen Ren
Hello from Zhen Ren
Hello from Zhen Ren
Hello from Zhen Ren
Hello from Zhen Ren
Hello from Zhen Ren
Hello from Zhen Ren
Hello from Zhen Ren
Hello from Zhen Ren
Hello from Zhen Ren
Process ../test1/user exited normally
```

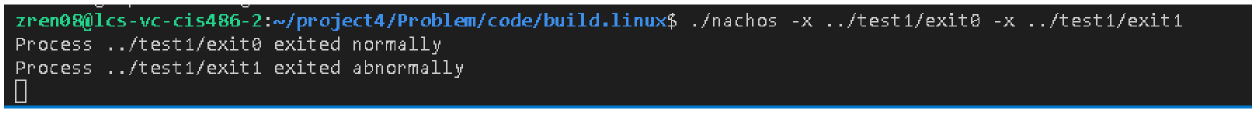
Firstly run user in SC_write put the hello from Zhen Ren into buf 10 times then output the whole buf in str format. Then call the SC_exit to output the process status, exited normally.

Q11:
exit0.c

```
#include "syscall.h"
int
main()
{
  Exit(0);
}
```

exit1.c

```
#include "syscall.h"
char data[2048];
int
main()
{
  Exit(1);
}
```



```
zren08@lcs-vc-cis486-2:~/project4/Problem/code/build.linux$ ./nachos -x ../test1/exit0 -x ../test1/exit1
Process ../test1/exit0 exited normally
Process ../test1/exit1 exited abnormally
```

Due to exit0 will exit(0), and exit1 will exit(1) so exit0 will output normally exit, exit1 will output abnormally exit.