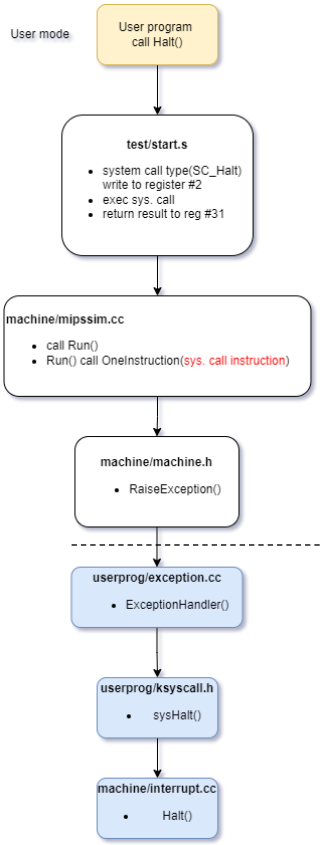
OS HW1 System call

Group 11:311512011 楊士賢

Part I: trace code Result

1. Flow Chart of Halt() System Call:

System Call: An interface provided by the operating system for user-level processes to request services or resources from the kernel.



Requesting Services: User programs request privileged operations, such as file I/O.

Context Switch: When a system call is made, the CPU switches from user mode to kernel mode, allowing privileged access.

Kernel Execution: The kernel performs the requested operation and returns results, like data or status information.

Return to User Space: After execution, the CPU switches back to user mode, and the program continues.

2. Details of Trace Halt() Code: halt the NachOS from executing

a. test/halt.c

```c
#include "syscall.h"

int
main()
{
    Halt();
    /* not reached */
}
```

Halt() is called in user program, , the NachOS operating system executes a corresponding stub

b. test/start.S: Acts as a bridge or intermediary between different parts of a program or different execution environments.

```
Halt:
    addiu $2,$0,SC_Halt
    syscall
    j   $31
    .end Halt

    .globl MSG
    .ent   MSG
```

- Write the type of the system call into register 2 (in this case, it is SC_Halt defined in system.h).
- Execute the system call instruction.
- Return to the address stored in register 31, which is the address of the user program.

c. machine/mipssim.cc Machine::Run

```c
void
Machine::Run()
{
    Instruction *instr = new Instruction;  // storage for decoded instruction

    if (debug->IsEnabled('m')) {
        cout << "Starting program in thread: " << kernel->currentThread->getName();
        cout << ", at time: " << kernel->stats->totalTicks << "\n";
    }
    kernel->interrupt->setStatus(UserMode);
    for (;;) {
        OneInstruction(instr);
        kernel->interrupt->OneTick();
        if (singleStep && (runUntilTime <= kernel->stats->totalTicks))
            Debugger();
    }
}
```

- When the system executes the syscall instruction, it is passed to Machine::Run() in mipssim.cc.
- The processor will pass the captured syscall instruction to execute in OneInstruction()

d. machine/mipssim.cc Machine::OneInstruction:

```c
  case OP_SYSCALL:
RaiseException(SyscallException, 0);
return;
```

- In the OneInstruction() function in mipssim.cc, the process simulates the execution of individual CPU instructions.
- when the processor encounters a syscall instruction, it invokes RaiseException(SyscallException, 0) to throw a SyscallException

e. RaiseException()

```
void
Machine::RaiseException(ExceptionType which, int badVAddr)
{
    DEBUG(dbgMach, "Exception: " << exceptionNames[which]);
    registers[BadVAddrReg] = badVAddr;
    DelayedLoad(0, 0);          // finish anything in progress
    kernel->interrupt->setStatus(SystemMode);
    ExceptionHandler(which);        // interrupts are enabled at this point
    kernel->interrupt->setStatus(UserMode);
}
```

- In RaiseException() passes the SyscallException to the ExceptionHandler() function.
- **kernel->interrupt->setStatus(SystemMode)** signifies the transition from User Mode to Kernel Mode at this point.
- **kernel->interrupt->setStatus(UserMode)** indicates that after ExceptionHandler() execution, the transition is made from Kernel Mode back to User Mode."

f. ExceptionHandler()

```
case SyscallException:
    switch(type) {
    case SC_Halt:
        DEBUG(dbgSys, "Shutdown, initiated by user program.\n");
        SysHalt();
                    cout<<"in exception\n";
        ASSERTNOTREACHED();
        break;
```

- This function first determines which type of exception is being passed into it.
- If it's a SyscallException, it will then determine, based on the type extracted from register 2, what action to take. It will execute the specific tasks associated with the system call type.
- For SC_Halt, the specific processing will be carried out in SysHalt()

g. SysHalt()

```
void SysHalt()
{
  kernel->interrupt->Halt();
}
```

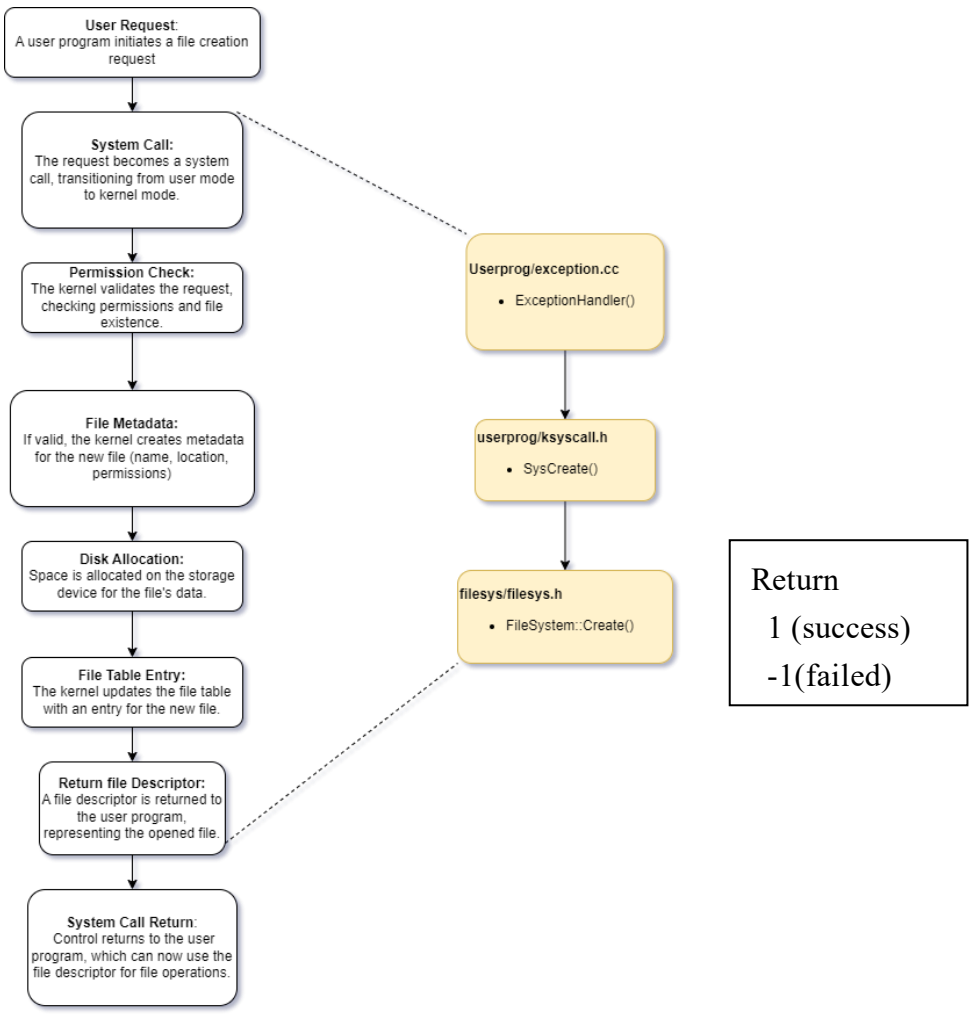
- Entering the SysHalt() function located in ksyscall.h, you can observe that it jumps to the Halt() function in interrupt.cc

h. Halt()

```
void
Interrupt::Halt()
{
    cout << "Machine halting!\n\n";
    cout << "This is halt\n";
    kernel->stats->Print();
    delete kernel;  // Never returns.
}
```

- delete the kernel: The kernel is responsible for linking all programs and functions, so deleting the kernel program will stops the system.

3. Flow Chart of Create() System Call:



4. Details of Trace Create() Code

a. userprog/exception.cc

```
case SC_Create:
    //read input argument from register #4
    val = kernel->machine->ReadRegister(4);
    {
    // get file name to be opened
    char *filename = &(kernel->machine->mainMemory[val]);

    //System call
    status = SysCreate(filename);

    //write result(status flag) to register #2
    kernel->machine->WriteRegister(2, status);
    }

    // Write PC
    kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));
    kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) + 4);
    kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCReg)+4);
    return;
    ASSERTNOTREACHED();
    break;
```

- Checks the exception type passed into the function.
- If it's a system call, it then uses the system call type stored in register 2 to determine and execute the corresponding actions for that system call type.

b. userprog/ksyscall.h

```
int SysCreate(char *filename)
{
    // return value
    // 1: success
    // 0: failed
    return kernel->interrupt->CreateFile(filename);
}
```

- Enter SysCreate() function located in ksyscall.h, it will call function CreateFile() in filesystem.cc

c. Filesys/filesys.cc

```
bool
FileSystem::Create(char *name, int initialSize)
{
    Directory *directory;
    PersistentBitmap *freeMap;
    FileHeader *hdr;
    int sector;
    bool success;

    DEBUG(dbgFile, "Creating file " << name << " size " << initialSize);

    directory = new Directory(NumDirEntries);
    directory->FetchFrom(directoryFile);

    if (directory->Find(name) != -1)
      success = FALSE;          // file is already in directory
    else {
        freeMap = new PersistentBitmap(freeMapFile,NumSectors);
        sector = freeMap->FindAndSet(); // find a sector to hold the file header
        if (sector == -1)
            success = FALSE;        // no free block for file header
        else if (!directory->Add(name, sector))
            success = FALSE;    // no space in directory
    else {
            hdr = new FileHeader;
        if (!hdr->Allocate(freeMap, initialSize))
                success = FALSE;    // no space on disk for data
        else {
            success = TRUE;
        // everthing worked, flush all changes back to disk
                hdr->WriteBack(sector);
                directory->WriteBack(directoryFile);
                freeMap->WriteBack(freeMapFile);
        }
            delete hdr;
    }
        delete freeMap;
    }
    delete directory;
    return success;
}
```

The function check the following,

- ✓ The file already exists in the folder.
- ✓ It is not possible to allocate available physical blocks for the new file's file header.
- ✓ There is not enough space in the folder to create a new file.
- ✓ The physical disk has run out of space to write data.

In all other cases not covered by these four scenarios, the function returns True, indicating that the file creation was successful.

5. Details of Makefile

```
include Makefile.dep

CC = $(GCCDIR)gcc
AS = $(GCCDIR)as
LD = $(GCCDIR)ld

INCDIR =-I../userprog -I../lib
CFLAGS = -G 0 -c $(INCDIR) -B../../usr/local/nachos/lib/gcc-lib/decstation-ultrix/2.95.2/ -B../../usr/local/nachos/decstation-ultrix/bin/

ifeq ($(hosttype),unknown)
PROGRAMS = unknownhost
else
# change this if you create a new test program!
# PROGRAMS = add halt consoleIO_test1 consoleIO_test2 fileIO_test1 fileIO_test2
PROGRAMS = add halt consoleIO_test1 consoleIO_test2 fileIO_test1 fileIO_test2
endif

all: $(PROGRAMS)

start.o: start.S ../userprog/syscall.h
    $(CC) $(CFLAGS) $(ASFLAGS) -c start.S

halt.o: halt.c
    $(CC) $(CFLAGS) -c halt.c       TA, 3 weeks ago • init: init nachos hw01, should pass jenkins os_gr…
halt: halt.o start.o
    $(LD) $(LDFLAGS) start.o halt.o -o halt.coff
    $(COFF2NOFF) halt.coff halt
```

1. include Makefile.dep: Includes another Makefile called "Makefile.dep." This is often used to include dependencies and rules defined in another Makefile.
2. Definitions of Compiler and Tools:
   - **CC = $(GCCDIR)gcc**: This defines the C compiler as **gcc** and prefixes it with **$(GCCDIR)**, which is likely a user-defined variable to specify the directory where the GCC (GNU Compiler Collection) tools are located.
   - **AS = $(GCCDIR)as:** This defines the assembler as **as**, which is used for assembly language programming.
   - **LD = $(GCCDIR)ld**: This defines the linker as **ld**, which is used to link object files into an executable.
3. Compilation and Linking Options:
   - **INCDIR = -I../userprog -I../lib**: This variable defines include directories for the C compiler. It tells the compiler where to look for header files.
   - **CFLAGS = -G 0 -c $(INCDIR) -B../../usr/local/nachos/lib/gcc-lib/decstation-ultrix/2.95.2/ -B../../usr/local/nachos/decstation-ultrix/bin/**: This variable defines C compiler flags. It specifies options like optimization level, include directories, and library search paths.
4. Conditional Block:
   - **ifeq ($(hosttype),unknown)**: Checks if the variable **hosttype** is equal to "unknown." If it is, it sets the variable **PROGRAMS** to "**unknownhost**," which likely means that the build process will create an executable named "**unknownhost**."
   - If **hosttype** is not "unknown," it sets **PROGRAMS** to a list of programs to

be built, including "**add**" and "**halt**."

5. Target Definition:
   - **all: $(PROGRAMS):** This defines a target called "all" that depends on the targets specified in the **PROGRAMS** variable. When you run make all, it will build the programs listed in **PROGRAMS**
6. Compilation Rules:
   - **start.o: start.S ../userprog/syscall.h:** This rule specifies how to build the object file start.o from the assembly file start.S and the header file **../userprog/syscall.h.** It uses the C compiler **($(CC))** and the flags defined in **CFLAGS**.
   - **halt.o: halt.c:** This rule specifies how to build the object file halt.o from the C source file halt.c. It also uses the C compiler and CFLAGS.
   - **halt: halt.o start.o**: This rule specifies how to build the executable file halt from the object files halt.o and start.o. It uses the linker ($(LD)) and likely linker flags ($(LDFLAGS)) that are not defined in this snippet. The result is the creation of halt.coff and then a conversion to halt using COFF2NOFF

```
clean:
    $(RM) -f *.o *.ii
    $(RM) -f *.coff

distclean: clean
    $(RM) -f $(PROGRAMS)

unknownhost:
    @echo Host type could not be determined.
    @echo make is terminating.
    @echo If you are on an MFCF machine, contact the instructor to report this problem
    @echo Otherwise, edit Makefile.dep and try again.
```

Clean:
$(RM) -f *.o *.ii: It uses the $(RM) variable (which is likely set to "rm") to remove all files with a .o extension and all files with a .ii extension in the current directory. $(RM) -f *.coff: It also removes all files with a .coff extension in the current directory.

Distclean:
$(RM) -f $(PROGRAMS): It uses the **$(RM)** variable to remove the files listed in the **PROGRAMS** variable. This effectively deletes the executable files generated by the build process.

Part 2：Implement System Call

1. Detail of your Console I/O system call implementation

A. Define SC_PrintInt macro

```
#define SC_PrintInt 16
```

B. Modify start.s

```
PrintInt:
    addiu $2,$0,SC_PrintInt
    syscall
    j   $31
    .end PrintInt

    .globl Add
    .ent    Add
```

C. Define function in ksyscall.h

```
void SysPrintInt(int num){
    if(!num){
        putchar('0');
        putchar('\n');
    }

    int temp = num;
    int i = 1;

    while(temp){
        i*=10;
        temp/=10;
    }
    i/=10;

    while(num){
        putchar(num/i + '0');
        num%=i;
        i/=10;
    }
    while(i--){
        putchar('0');
    }
    putchar('\n');
}
```

After get the integer from user, change every digit into character, then use the putchar() to print the result to console.

2. Detail of your File I/O system call implementation

A. Define macro for four different operations. Later will be used to determined which operation is being call.

```
#define SC_Open      6
#define SC_Read      7
#define SC_Write     8
#define SC_Close     10
```

B. Modify start.s

Read the calue from register #2 and call the corresponding syscall
Register #4, #5, #6, and #7 store the four parameters $a0, $a1, $a2, and $a3

```
Open:
    addiu $2,$0,SC_Open
    syscall
    j   $31
    .end Open

    .globl Read
    .ent    Read
Read:
    addiu $2,$0,SC_Read
    syscall
    j   $31
    .end Read

    .globl Write
    .ent    Write
Write:
    addiu $2,$0,SC_Write
    syscall
    j   $31
    .end Write

    .globl Close
    .ent    Close
Close:
    addiu $2,$0,SC_Close
    syscall
    j   $31
    .end Close

    .globl Seek
    .ent    Seek
```

C. **cxception.cc**

```
case SC_Open:
{
    val = kernel->machine->ReadRegister(4);

    char *file_name = &(kernel->machine->mainMemory[val]);
    fid = SysOpen(file_name);

    kernel->machine->WriteRegister(2, fid);

    // Write PC
    kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));
    kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) + 4);
    kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCReg)+4);
    return;
    ASSERTNOTREACHED();
    break;
}
```

SC_Open: Extract the argument from register r4, which is the file name's location in main memory. Retrieve the file name from main memory and call SysOpen(filename). The fid will receive either the file ID or -1, indicating success or failure in opening the file.

```
case SC_Close:
{
    val = kernel->machine->ReadRegister(4);

    int ret = SysClose(val);
          You, 19 hours ago • fix exception.cc
    kernel->machine->WriteRegister(2, ret);

    // Write PC
    kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));
    kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) + 4);
    kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCReg)+4);
    return;
    ASSERTNOTREACHED();
    break;
}
```

SC_Close: Extract the file ID from register r4, which is the only argument for closing the file. Call SysClose(fileID). The ret will receive 0 for failure to close the file or 1 for successful file closure.

```
case SC_Read:
{
    val = kernel->machine->ReadRegister(4);   // first argument
    size = kernel->machine->ReadRegister(5);  // second argument
    fid = kernel->machine->ReadRegister(6);

    char *data_read = &(kernel->machine->mainMemory[val]);

    int num_read = SysRead(data_read, size, fid);

    kernel->machine->WriteRegister(2, num_read);

    // Write PC
    kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));
    kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) + 4);
    kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCReg)+4);
    return;
    ASSERTNOTREACHED();
    break;
}
```

SC_Read: Similar to SC_Write, but the final call is SysRead(buffer, size, fileID). The num_read receives the size of the data read.

```
case SC_Write:
{
    val = kernel->machine->ReadRegister(4);   // first argument
    size = kernel->machine->ReadRegister(5);  // second argument
    fid = kernel->machine->ReadRegister(6);

    char *data_write = &(kernel->machine->mainMemory[val]);

    int num_write = SysWrite(data_write, size, fid); //return num of write

    kernel->machine->WriteRegister(2, num_write);

    // Write PC
    kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));
    kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) + 4);
    kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCReg)+4);
    return;
    ASSERTNOTREACHED();
    break;
}
```

SC_Write: Extract the following from registers r4 to r6, respectively, the location of the content to be written in main memory, the size of data to be written, and the file ID to write to. Call SysWrite(buffer, size, fileID). The num_write variable will receive the size of the data that was written.

D. ksyscall.h

```
OpenFileId SysOpen(char *file_name){
    return kernel->fileSystem->Open_File(file_name);
}

int SysWrite(char *buffer, int size, OpenFileId id){
    return kernel->fileSystem->Write_File(buffer, size, id);
}

int SysRead(char *buffer, int size, OpenFileId id){
    return kernel->fileSystem->Read_File(buffer, size, id);
}

int SysClose(OpenFileId id){
    return kernel->fileSystem->Close_File(id);
}
```

Continuing from the ExceptionHandler, it calls the SysOpen() function defined in ksyscall.h. This indirect call is a way to interact with the kernel to perform file-related operations.

E. filesys.h

In filesys.h we implement four different operation using the function pre-defined in sysdep.cc and return the result back.

```c
OpenFileId Open_File(char *name){
  int fileDescriptor = OpenForReadWrite(name, FALSE);
    return fileDescriptor;
}

  int Write_File(char *buffer, int size, OpenFileId id){
    WriteFile(id, buffer, size);
    return size;
  }

  int Read_File(char *buffer, int size, OpenFileId id){
    Read(id, buffer, size);
    return size;
  }

  int Close_File(OpenFileId id){
    int ret = Close(id);
    return ret >= 0 ? 1: -1;
}
```