國立清華大學
NATIONAL TSING HUA UNIVERSITY

Department of Electrical Engineering and Computer Science

CS342300 - Spring 2023

MP1 – System Call

Annabella Putri Dirgo
109006238

1 Trace Code NachOS System Call

1.1 SC_Halt

The operating system responds to a "halt" system call by instantly terminating the program's execution and shutting down the system.

Our tracing starts from `machine/mipssim.cc` which serves as a part of machine emulation which means it is responsible to fetch instruction or execute a command

When the machine wants to generate an interrupt, it will run the function `machine::run` and `machine::OneInstruction`. To shutdown the system, it will raise an exception in `machine::RaiseException`.

In that exception, it will be handle by the function `exceptionhandler()` in `userprog/exception.cc`. Inside that exception, we will see what kind of interrupt that we want to run. Since we want to run Halt we will switch into case `SC_Halt`.

Inside the case function there is a function call named `sysHalt()` in `userprog/ksyscall.h`. Thus, if we jump to the function it will generate an interrupt to do Halt plus if we see in `machine/interrupt.cc` it will perform halt and the machine will successfully shutdown.

1.2 SC_Create

When we want to perform `SC_Create`, in file `userprog/exception.cc`, it wil go through the function `exceptionhandler()` and find the exception type to perform create. Next, we head to the function `SC_Create()` and inside that function, it will prepare an amount of register to create a file and we took its file name and do a function call of `sysCreate` and pass its file name. Last, we jump to `filesys.h` and excute file creation.

1.3 SC_PrintInt

Since we want to perform `SC_PrintInt`, in `userprog/exception.cc` it will read the function `ExceptionHandler()` and read the case `SC_PrintInt`

First, it read the value of a register in a machine's kernel and store it to `val` and we call the function`SysPrintInt` from `userprog/ksyscall.h` and pass `val` .

Inside the the `SysPrintInt` function, the function will calls `PutInt()` which is a method of the `synchConsoleOut` object to print the value to the console output. The `synchConsoleOut` are located in `userprog/synchconsole.cc`.

To write a character to the console display, we head to `userprog/synchconsole.cc` and find two functions that serve the purpose of writing, namely `PutChar(char ch)` and `PutInt(int value)`.

Since the value `val` is in type `int`, thus it will went through the function`PutInt(int value)` first and through this line of code `consoleOutput->PutChar(str[idx]);` it will convert into`char` through this function `PutChar(char ch)` and write a character to the console display.

Next, it will also run `ConsoleOutput::PutChar()` which ensures that characters are output to the console in a controlled manner, preventing multiple characters from being output at the same time and ensuring that the console output is scheduled properly. To conduct it, the machine will launch an interrupt in `Interupt::Schedule()`. This function in charge of scheduling interruptions for certain times in the future. The operating system may guarantee that interrupts are performed in the proper sequence and at the correct times by utilizing the pending list to keep track of pending interrupts and inserting new interrupts into the list in sorted order.

After receive the instruction, the machine will run using the function `Machine::Run()`. a "tick" refers to a single unit of time used by the simulator to keep track of the progress of a program. The simulator increments a global tick counter `kernel->stats->totalTicks` every time it executes an instruction or processes an interrupt.

While it run, the machine will consistently check whether there is any interrupt that are scheduled. This will be run in `Interrupt::CheckIfDue` with a return of bool. Hence, if there is a character input the machine will call `ConcolseOutput::Callback()` and at last it will run `synchConsoleOutput::Callback()`.

2 Implementation

2.1 OpenFileId Open(char *name)

```
OpenFileId OpenAFile(char *name) {
    OpenFile *file = Open(name);
    if (file == NULL) return -1;
    for (int i = 0; i <= 20; i++){
```

```
    if (OpenFileTable[i] == NULL){
      OpenFileTable[i] = file;
      return i;
    }
  }
  return -1;
}
```

The `OpenAFile` accept an argument with the reference to a null-terminated string `name` and returns the index its corresponding `OpenFileId`, which is an integer identifier for an open file.

First, I call the function`Open` and passed the parameter of `name` to a pointer `file`. If the file is empty then it will return -1. Otherwise, it will run a for loop to find a slot in the open file table for this file and if there is a slot it will assign the file then return the index of the file. At last, I also handle a case when when the table is full, it will return an error code of -1.

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

```
int WriteFile(char *buffer, int size, OpenFileId id) {
  if (buffer == NULL || size < 0) return -1;
  if (id < 0 || id > 20 || OpenFileTable[id] == NULL)return -1;
  int status = OpenFileTable[id]->Write(buffer, size);
  if(status < 0) return -1;
  return status;
}
```

The `WriteFile` function accepts a buffer pointer, an integer size, and an `OpenFileId` as parameters and returns an integer.

First, I handle a case when `buffer` is empty or the input of the `size` is not valid then it will raise an error. Moreover, If the `OpenFileId`is invalid (less than 0 or higher than 20), the method returns -1. Instead, If the id is valid, set status to the return value of the Write function, passing in the buffer and size. The function saves the return value of this call in a variable called status which returns the file size.

2.3 int Read(char *buffer, int size, OpenFileId id);

```
int ReadFile(char *buffer, int size, OpenFileId id) {
  if (buffer == NULL || size < 0) return -1;
  if (id < 0 || id > 20 || OpenFileTable[id] == NULL)return -1;
  int status = OpenFileTable[id]->Read(buffer, size);
  if(status < 0) return -1;
  return status;
}
```

The `ReadFile` function accepts a buffer pointer, an integer size, and an `OpenFileId` as parameters and returns an integer.

First, I handle a case when `buffer` is empty or the input of the `size` is not valid then it will raise an error. Moreover, If the `OpenFileId`is invalid (less than 0 or higher than 20), the method returns -1. Instead, If the id is valid, set status to the return value of the Read function, passing in the buffer and size. The function saves the return value of this call in a variable called status which returns the file size.

2.4 int Close(OpenFileId id);

```
int CloseFile(OpenFileId id) {
  if (id >= 0 && id < 20 && OpenFileTable[id] != NULL) {
      delete OpenFileTable[id];
      OpenFileTable[id] = NULL;
      return 1;
  }
  return -1;
}
```

The `CloseFile` function accepts an `OpenFileId` as an input and returns an integer. If the `OpenFileId` is invalid, that is, less than 0 or higher than 20, the method returns -1. Otherwise, the functions frees up he

memory by calling `delete` on the entry of `OpenFileTable` array that corresponds to the object. Next, I set the `OpenFileTable` into `NULL` whereby the file is no longer open. If the function is unable to close the file, it returns -1 to signal an error.

2.5 Difficulties

Several difficulties that I encounter while learning to trace code in Nachos system calls

- the code is low-level and interacts directly with the Nachos kernel, which means you need to know about operating systems and the architecture of Nachos.
- the code uses a lot of different C++ programming techniques, such as pointers, memory management, and class inheritance, which can be challenging and time consuming to understand.
- the Nachos codebase is large and complex, making it difficult to navigate and understand the various components and their interactions.