CS 691 Project Report

on

Guest OS implementation

By -
Prajakta Ayachit (09305924)
Roll NO-09305924

Guided By
Prof. D. M. Dhamdhere
Indian Institute of Technology,
Bombay

8 May 2011

Contents

Chapter 1

Introduction and Motivation

1.1 Motivation

For courses in Operating Systems,it is very important that students should understand the operating system concepts correctly. However assignments or projects designed to cater this should not consume lot of time of the students[**?**]. To strike a balance of time vs effective understanding an instructor can select to vary the "depth" of these assignments. e.g. As two extremes we can consider depth=OS source code and depth= simulation.

- "Simulation" of selected functionalities of operating systems: For this depth, some OS functionalities are simulated and students use simulation to study operational properties.

  Advantages: Easy to do and lesser amount of time required to dedicate
  Disadvantages:

  - Purpose of studying policies is achieved but not their implementation in OS
  - Design and coding of those policies in real operating systems is not touched

- Modifying parts of full-fleged OS: In this approach the student is supposed to make changes in code of real operating system (may be productional e.g. Unix or instructional e.g. OS/161)

  Advantages: Student can understand the actual design and implementation of real OS feature
  Disadvantages:

  - Time consuming
  - If a student fails to do earlier assignment,it may affect later assignments

This is the motivation of *Variable depth OS workbench*[**?**] . With the help of this workbench an instructor can schedule different OS assignments at different depths.

1.2 Introduction

Guest OS can be looked at as a part of "Variable depth OS workbench" . The OS that works under the workbench is called a *Guest OS* to differentiate it from Host OS.

This guest OS will act as OS and will have the capacity to run user processes which are C program executables.It will provide certain basic functionalities e.g scheduler,virtual memory ,interrupt handling,etc. The writing of Guest OS under the workbench would expose the student to the fundamentals of

- Booting of OS
- Handling of interrupts
- Implementation of system calls
- Working of signals

1.3 What is Guest OS?

[**?**] The guest OS itself is a C program. The user programs are the C programs compiled under GCC. There is one x86 simulator which will have certain features and will run under host OS as a process. This "guest OS + user process" will in turn run under the simulator. To execute the guest OS, the x-86 simulator will take object form of guest os and will simulate one machine instruction of guest OS at a time.

For host OS this combination of simulator + guest OS + user program will be single process and is called as Guest OS process here onwards.

Chapter 2

System Architecture

2.1 Components

The entire system can be seen to be composed of following components:

- Host OS: The Operating system on which the system runs
- X86 Simulator: The the x86 simulator capable of simulating instructions
- Guest Os: The OS of workbench running on top of simulator
- User process: The user process runs on top of guest OS and is executable program

2.2 Architecture

As shown in the figure 2.1[**?**] , the simulator runs as a process on host OS. The guest OS runs on top of simulator and user process runs on top of guest OS.The simlator has its memory called as *simulated memory.* In this simulated memory the guest OS is loaded.The guet OS then has following functionalities.

2.2.1 Booting of Guest OS

The booting code[**?**] of guest OS prepares it for operation. The simulator is given guest OS to execute as its parameter. After guest OS is loaded in simulated memory it gives calls to installer routines. These installer routines are:- system_call_installer, signal_installer , interrupt_installer. These routines take number of syscall, or signal_no or interrupt_no (respectively) and address of the handler routine as parameter. e.g. install_syscall_handler would add a syscall call handler of given system call in syscall vector.

Apart from booting code,the guest OS requires loader to load the user process in its memory. Also data stucture initialization of virtual memory (paging etc where data structures e.g. paga table) takes place.

The installer routines are discussed in detail in 4.

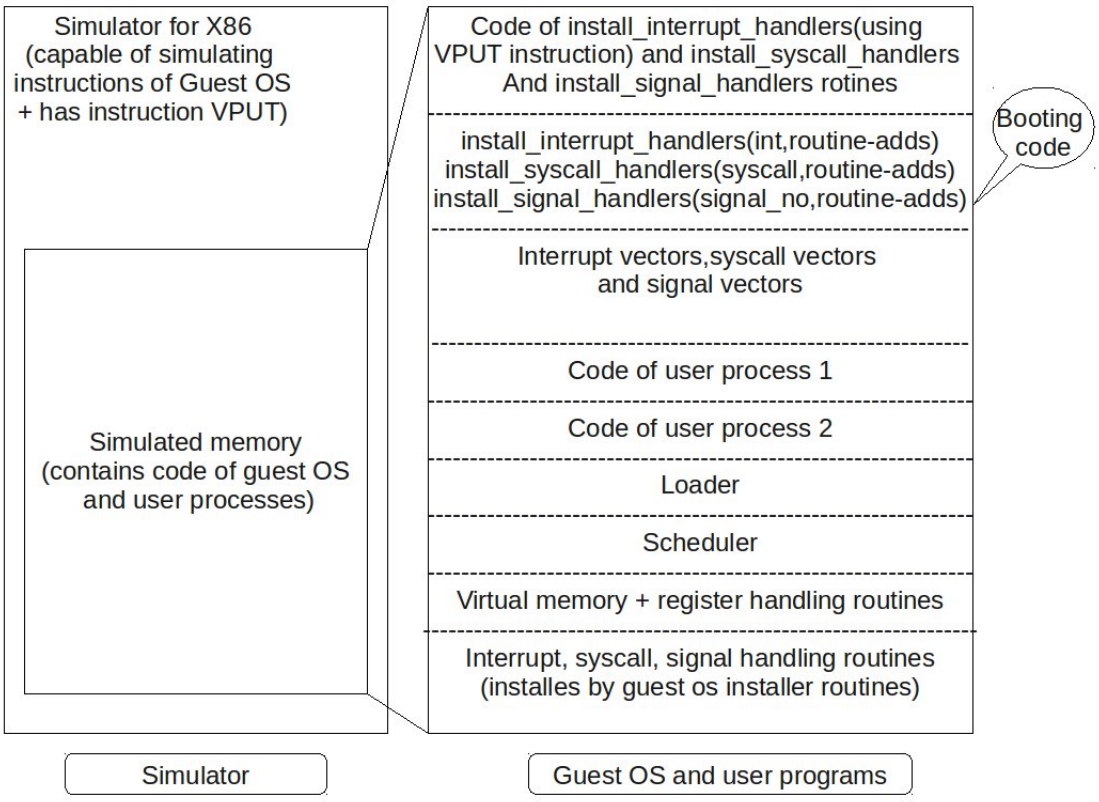

Figure 2.1: System architecture

Chapter 3

x86 Simulator

3.1 Required Features

In the entire guest OS system, the x86 simulator plays a key role. It simulates functioning of an x86 processor at desired level of abstraction. For this it should have following features:

- It should be able to simulate all commonly used instructions of x86 processor.
- Features such as memory mangement unit(MMU), page tables ,translation lookaside buffer (TLB), direct memory access (DMA) for I/O. It may support caching at functional level.
- Interrupts
- System calls
- Extension to instruction x86 set: It should be able to modify the instruction set of x86 processor to support one additional instruction called "VPUT".This instruction will mainly be used in case of interrupt handling. The purpose of instruction vput(address,value) will be to put given value (second parametr) at the given address(first parameter) of simulated memory.
- Provision for interfacing with Host OS: This feature will be useful for system call handling. Using this, some system calls will be passed on to host OS. The details of this are discussed in 4.

3.2 Simulators tried

As a part of this RnD ,I have tried few simulators. Their information in short is as follows:

- Bochs: This is a very famous emulator. This expects as input an image of operating system to be run. Here the input to be emulated should have all the features

including loader, MMU ,I/O handling and all implemented. In this guest OS project ,the focus is that the students should understand the concept and make changes in exiting structure and see the effect. Bochs was not used further as it would have required to implement everything on own as in a basic operating system such as Minix. So here lot of time will be spent on things which are not very useful.
URL :`http://bochs.sourceforge.net/`

- SimpleScalar:- This is a simulator for various architectures such as x86, Alpha, MIPS, etc. It is capable of running simple C program. However there were two problems:
  - Difficult to install:- Lot of bugs need to be fixed while installation
  - GCC 2.7.2.3 issue :- When this simulator was very famous around 1990's , GCC had provided a support for it. To run a C program under simplescalar, it is required that it should be compiled under GCC 2.7.2. This was the last version of GCC which supported SimpleScalar. There was some problem with automatic code generation(wrong opcode in machine definition file)of GCC version. For these two problems this simulator was not continued further.
    URL: `http://www.simplescalar.com/`
- Sim-Zesto:- This x86 simulator which can support multi programming as well as multi core programs.Zesto is a timing model constructed based on the pre-release version of the x86 version of SimpleScalar. This simulator was developed for detailed cycle-level modeling of modern out-of-order pipelines at a level of detail much greater than is typical of academic pipeline simulators. However it was not continued further due to some implementation problems.
  URL: `http://zesto.cc.gatech.edu/`
- Multi2sim:- This is the current simulator being used for RnD project. Multi2Sim is a simulation framework for state-of-the-art processors with the following features[**?**]:-
  - Superscalar pipeline:-Out-of-order execution, branch prediction, trace cache, etc.
  - Multithreading:-Fine-grain, coarse-grain and simultaneous (SMT).
  - Multicore architecture:-Configurable memory hierarchy, cache coherence, interconnection networks.
  - Quick setup and test:-Easy to install
  - GPU model(in progress) :-Simulation of OpenCL programs targeting AMD Evergreen ISA.

  The details of some of the modules of the simulator are discussed in next section.
  URL: `http://www.multi2sim.org/wiki/index.php5/Main_Page`

3.3 Multi2sim modules

This section describes some of the multi2sim modules in short[**?**]:-

- Cache system:- cahe policies supported are LRU, FIFO, Random. The related files [containing functions to create cache, access data, update cache, implement cache policy] are guestos_multi2sim/src/libcachesystem/cachesystem.c and guestos_multi2sim/src/libcachesystem/cahe.c .

- Core files:- These include functionalities of loading, register handling, signal and system call processing.

  - Context.c :- This file is mainly used for creating contexts of processes. Here context is more than PCB. It involves many things about a process including its loader, signal handler list, segment bases, segment limits ,memory details, file descriptor table ,etc. The file has functions to create and destroy contexts for processes. One context is created per process. Also for handling children processes cloning of context is done.
  - elf.c :- This file is used for loading purpose. Functions related to elf loader handling are found here. These are called by functions from file loader.c
  - fs.c :- File related to file sytem. But they have not actually implemented file system of their own. Only some data structures are maintained to keep track of files opened, created , accessed.
  - isa.c and machine.c :- These file together handle things at machine level. e.g. register mapping, returning immediate operand from instruction, handling loads and stores, maintaining instruction statistics, etc.
  - loader.c :- File required to load a user inputted executable to load into simulated memory. ELF loading format is followed. Headers are checked and appropriate sections are allocated memory and loaded.
  - m2skernel.c :- This file calls functions from other files discussed above and below.It keeps track of timing for simulation. It runs processes from each context one by one, processes the events.
  - memory.c :- This has virtual memory related functions implemented. This file has functions doing task of memory mapping, address translation, allocation and freeing of required number of pages ,etc.

- Instruction cycle stages :- The instruction cycle stages are implemented in /guestos_multi2sim/src/fetch.c , /guestos_multi2sim/src/decode.c, /guestos_multi2sim/src/issue.c , /guestos_multi2sim/src/commit.c.

- Others :- There are files for disasembler, events handling, branch prediction, mutiprocessesing ,etc

To summarize Simulator provides basic structure to load and execute C program. Hence it has functinalities such as fetch, decode, execute, etc. It has some support for signal and system call handling, MMU. However there is no interrupt handling and I/O handling present as per requirement.

Chapter 4

Guest OS added functionalities

4.1 Introduction

In this Rnd project I have implemented certain functionalities in Guest OS. This chapter mainly discusses system call handling, signal handling and design of interrupt handling and scheduling.

4.1.1 System call handling:

System call handling in real OS

Basically a system call [**?**]is how a user program requests a service from an operating system's kernel that it does not normally have permission to run. System calls provide the interface between a process and the operating system. The system calls are functions used in the kernel itself. To the programmer, the system call appears as a normal C function call. However since a system call executes code in the kernel, there must be a mechanism to change the mode of a process from user mode to kernel mode. The C compiler uses a predefined library of functions (the C library) that have the names of the system calls. The library functions typically invoke an instruction that changes the process execution mode to kernel mode and causes the kernel to start executing code for system calls. Usually for older Unix systems this instruction is "int 0x80". Also now-a-days same effect at a faster speed is achieved by executing the sysenter [**?**]assembly language instruction, introduced in the Intel Pentium II microprocessors (this instruction is now supported by the Linux 2.6 kernel) and sysexit to go back. So the control flow becomes
User mode(System call invocation in application program → wrapper routine in libC standard library) → kernel mode(generic system call handler calling sys_xyz → system call service routine).

The system call handler performs the following operations:[**?**]

- Saves the contents of most registers in the Kernel Mode stack (this operation is common to all system calls and is coded in assembly language).

- Handles the system call by invoking a corresponding C function called the system call service routine.
- Exits from the handler: the registers are loaded with the values saved in the Kernel Mode stack, and the CPU is switched back from Kernel Mode to User Mode (this operation is common to all system calls and is coded in assembly language).

The corresponding relevant code in short can be seen as [**?**]:

```
/* include/asm-i386/hw_irq.h */
#define SYSCALL_VECTOR              0x80

/* arch/i386/kernel/traps.c */
        set_system_gate(SYSCALL_VECTOR,&system_call);

/* arch/i386/kernel/entry.S */
#define GET_CURRENT(reg) \
        movl $-8192, reg; \
        andl %esp, reg

#define SAVE_ALL \
        cld; \
        pushl %es; \
        pushl %ds; \
        pushl %eax; \
        pushl %ebp; \
        pushl %edi; \
        pushl %esi; \
        pushl %edx; \
        pushl %ecx; \
        pushl %ebx; \
        movl $(__KERNEL_DS),%edx; \
        movl %edx,%ds; \
        movl %edx,%es;

#define RESTORE_ALL     \
        popl %ebx;      \
        popl %ecx;      \
        popl %edx;      \
        popl %esi;      \
        popl %edi;      \
        popl %ebp;      \
        popl %eax;      \
1:      popl %ds;       \
2:      popl %es;       \
        addl $4,%esp;   \
3:      iret;
```

```
ENTRY(system_call)
        pushl %eax                      # save orig_eax
        SAVE_ALL
        GET_CURRENT(%ebx)
        testb $0x02,tsk_ptrace(%ebx)    # PT_TRACESYS
        jne tracesys
        cmpl $(NR_syscalls),%eax
        jae badsys
        call *SYMBOL_NAME(sys_call_table)(,%eax,4)
        movl %eax,EAX(%esp)             # save the return value

ENTRY(ret_from_sys_call)
        cli                             # need_resched and signals atomic test
        cmpl $0,need_resched(%ebx)
        jne reschedule
        cmpl $0,sigpending(%ebx)
        jne signal_return
        RESTORE_ALL
```

There are approximately 300 system calls in kernel version 2.6.32[**?**]. When a system call is considered, depending on architecture and OS parameters are passed. For system call parameters are written in the CPU registers before issuing the system call[**?**]. The kernel then copies the parameters stored in the CPU registers onto the Kernel Mode stack before invoking the system call service routine, Register *eax* is used for passing system call number. So the registers used to store the system call number and its parameters are, in increasing order, *eax* (for the system call number), *ebx,ecx, edx, esi, edi* and *ebp*. The system_call( ) and sysenter_entry( ) save the values of these registers on the Kernel Mode stack by using the SAVE_ALL macro. Therefore, when the system call service routine goes to the stack, it finds the return address to system_call( ) or to sysenter_entry( ), followed by the parameter stored in ebx (the first parameter of the system call), the parameter stored in ecx, and so on.

However, there are system calls that require more than six parameters. In such cases, a single register is used to point to a memory area in the process address space that contains the parameter values.

Guest OS system calls

To add system call functionality to guest OS , the purpose was to make students understand system calls in an easy way. As part of RnD, system call handling can be divided in two parts. System calls are identified by their sytem call number. So the design is such that system calls in range say 1-325 are passed on to host OS and system calls in a range say 400-410 are added by guest OS and handled by guest OS.The range of guest OS handled system calls can be easily increased to a desired number by adding their entries in files syscall.c and syscall.dat as explained in implementation subsection.

- System calls with system call number <= 325 :- These system calls may be having less than or greater than 6 parameters. They are passed to the host OS. Before passing the system call, there is one problem with parameter handling. If a parameter is address of some location in user program a care needs to be taken.(e.g. fopen-string storing path of file to open). In such cases, the address is translated by MMU and after retriving the parameter , the parameter is copied to guestOS memory and then again the real system call is made.
  Also as the system call range is standard, there is no change in the way the user program performs a system call.

- System calls with system call number in the range 400-410:- These are the system calls which are added by guest OS. The range 400-410 is taken so as not to interfere with standard OS system calls. These system calls may not provide very big functionality . They are added to understand the system call concepts. So they perform local operations. Also as the range is different, if a user program wants to do a system call, it should be done by calling "syscall(syscall_no)" instruction from a user program.

Implementation

The files related to system call handling can be found /src/libm2skernel/syscall.c and /src/libm2skernel/syscall.dat . The syscall.c file contains all the necessary functions and data structures for system call handling. The sycall.dat file has syscall_number to syscall_name mapping. The function syscall_do() is called when the opcode of decoded instruction is same as op_int_imm8. This function is equivalent to generic syscall handler of host OS. This function further has switch call whose each case acts as syscall specific handler.

The relevant part of code from syscall.c and syscall.dat in short is as follows:

```
/* Usually, glibc wrappers to system calls return -1 on error and set the
 * 'errno' variable to the appropriate value. However, the ABI with the
 * operating system is different - a return value less than 0 specifies
 * the error code. We use this macro for this kind of system calls. */
#define RETVAL(X) { retval = (X); if (retval == -1) retval = -errno; }
//unsigned int myarr[10];

/*  Following are the functions which implement systemcalls provided
 by guest os*/

int guestos1()// for system call no 400
{
printf("\n Inside sytem call Number:400\n\n");
        printf("\n ****This is the first system call provided by guest os***");
printf("\n This system call does not have any other parameter
                                        other than sytem call number");
        printf("\n hello to world!!!!!!!");
return 0;
```

```
}
int guestos2(uint32_t ipstring) // For system call number 401
{
printf("\n Inside system call number:401\n\n");
char  strtoprint[80];
.
.
.
.
}
/* Simulation of system calls.
 * The system call code is in eax.
 * The parameters are given in ebx, ecx, edx, esi, edi, ebp.
 * The return value is placed in eax. */
void handle_guest_syscalls()
{
int syscode = isa_regs->eax;
int retval = 0;
switch(syscode)
{
.
.
.
.}
void syscall_do()
{
int syscode = isa_regs->eax;
int retval = 0;
if (syscode > 325)
{
handle_guest_syscalls();
}
else // Pass system call to host os
{
.
.
.
.
}
SYSCALL.dat file contains:-
DEFSYSCALL(restart_syscall, 0)
DEFSYSCALL(exit, 1)
DEFSYSCALL(fork, 2)
DEFSYSCALL(read, 3)
.
```

```
.
.
DEFSYSCALL(guestos1, 400)
DEFSYSCALL(guestos2,401)
DEFSYSCALL(guestos3,402)
```

The switch case in which there is call to system call handler from syscall.c is as :

```
  case syscall_code_guestos1:
{
  retval = guestos1();
  break;
}

case syscall_code_guestos2:
{
  uint32_t ipstring = isa_regs->ebx;
  retval = guestos2(ipstring);
  break;
}

case syscall_code_guestos3:
{
  retval = guestos3();
  break;
}

case syscall_code_guestos4:
{
  uint32_t path = isa_regs->ebx;
  retval = guestos4(path);
  //retval=guestos4(path);
  /* Return */
  break;
}

case syscall_code_guestos5:
{
  retval = guestos5();
  break;
}
.
.
.
default:
if (syscode >= syscall_code_count) {
```

```
retval = -38;
} else {
fatal("not implemented system call '%s' (code %d) at 0x%x\n%s",
syscode < syscall_code_count ? syscall_name[syscode] : "",
syscode, isa_regs->eip, err_syscall_note);
}
```

To deal with new system calls, a function named handle_guest_syscalls() is provided. When a user wants to add new system call,he needs to write a syscall handling routine here. There are currently 5 new system calls supported as examples.Their functionalities are:-

- syscall guest_os1 : It just prints a "hello" message to console. No parameters apart from syscall number.
- syscall guest_os2: It prints a user supplied message to console. Here the message is sent as a parameter by user program. Since message can be char[ ] or char * it is treated as address in user space.
- syscall guest_os3: It performs "ls" command on current directory. No parameter apart from syscall number.
- syscall guest_os4: It performs "ls " command on user supplied path. Again the path is taken as parameter and treated as address.

So as far as guest OS project is concerned, control flow is like :
User mode(System call invocation in application program → wrapper routine in libC standard library) →guest OS (system call handler syscall_do() → system call service routine switch case depending on syscall number).

4.1.2 Signal handling

Introduction

A signal is a limited form of inter-process communication used in Unix operating system. It is an asynchronous notification sent to a process in order to notify it of an event that occurred. When a signal is sent to a process, the operating system interrupts the process's normal flow of execution. Execution can be interrupted during any non-atomic instruction. If the process has previously registered a signal handler, that routine is executed. Otherwise the default signal handler is executed.
Signals can be initiated by :-

- Other processes:-e.g. parent process using kill command to kill child process
- The process itself:-This includes hardware exceptions triggered by a process e.g. when a program executes an illegal instruction, such as dividing a number by zero or attempting to access a memory zone that has not been allocated yet, the hardware detects it and a signal is sent to the faulty program.
- The Kernel:- e.g. alarm() syscall will send a signal to process evry time a timer expires.

Signal handling in real OS

[**?**] The Kernel generates a signal for a process simply by setting a flag that indicates the type of the signal that was received. More precisely, each process has a dedicated bitfield used to store pending signals; For the system, generating a signal is just a matter of updating the bit corresponding to the signal type in this bitfield structure. At this stage, the signal is said to be pending.
Before transferring control back to a process in user mode, the Kernel always checks the pending signals for this process. This check must happen in Kernel space because some signals can never be ignored by a process namely SIGSTOP and SIGKILL .

When a pending signal is detected by the Kernel, the system will deliver the signal by performing necessary action. e.g. terminating user process (in case of SIGKILL) or else if a user process registers a signal handler with the OS then on arrival of that signal the corrosponding signal handler is invoked. Otherwise default signal handler can be called if signal is not ignored. So a process can chose to -ignore signal,handle it itself or handle in a default way.

Most UNIX systems define about 30 signals[**?**]. Their action is dependent on which signal comes.

Signal handling in Guest OS

Signal handling is implemented in Guest OS in a similar way. It also provides pending signal checking and other behaviour described above. To understand the signal handling concept, guest OS provides its own signal handlers. So when a user process causes a signal, these signal handlers are executed if user process has not provided its own handler. If guest OS has not provided the handler then the real OS handler is invoked.

Implementation details

The signal handling related files can be found in /src/libm2skernel/signal.c and /src/libm2skernel/m2skernel.h . The file signal.c has a structure signal_map providing signal number to signal name mapping. The related functions such as sim_sigset_add() to add a signalto process's signal set and sim_sigset_del() to delete a signal after processing it etc are there in same file. One important structure is sim_sigframe. This structure has fields such as pointer to return code,received signal,signal mask etc. For handling signals ,signal_handlers are created. Signal_handler_run() function copies appropriate values to this structure ,saves it and then calls the signal respective handler.

To register the handler of guest OS, function pointers are used. install_signal_handlers() routine creates array of signal handlers and registers signal handlers. To have a new signal handler of guest OS, first the signal handler routine should be written. Then to register it , initialization of signal handlers array's i^{th} location by the handler routine's address should be done inside the install_signal_handlers() routine. Whenever a process is created i.e. its context is created, signal handler array is created as a part of it and hence the handler becomes effective.

4.1.3 Interrupt Handling

Overview

An interrupt [**?**] is an event that alters the sequence in which the processor executes instructions. A *hardware interrupt* causes the processor to save its state of execution and begin execution of an interrupt handler. *Software interrupts* are usually implemented as instructions in the instruction set, which cause a context switch to an interrupt handler similar to a hardware interrupt.

Interrupts can be maskable, non-maskable(NMI) or software interrupt[**?**]. For handling interrupts, handler routines are present which are called *Interrupt Service Routines.*

Interrupt handling in OS

[**?**] In real OS, a data structure called Interrupt vector table is present. This table has interrupt number and handler routine address. Accessing ISR(interrupt service routine) takes place via gates :- interrupt gates, task gates, trap gates. They have different flags which decide what kind of interrupt it is and who can access it. The interrupt descriptor's privilege level(DPL) is compared with user's current privilege level (CPL) and then the access is granted.

Guest OS interrupt handling

The simulator multi2sim has no support for interrupt handling. So currently there is no interrupt handling supported in guest OS. Ideally if the support is there, install_interrupt_handler(interrupt_no,address) should install the routine whose address is given as parameter in IVT for corrosponding "interrupt_no" numbered interrupt.

For this it will require one instruction called VPUT. This instruction(address,val) will put the routines address given by second parameter at a location "address" of simulated memory. The location in this case will be the*base address of IVT + 8 * interrupt_no* This way a user will be able to provide its own handler for interrupt.

4.1.4 Scheduler

The scheduler of any OS decides which process to schedule at given instant of time. For this purpose, scheduler can use its own scheduling policy.

Design of scheduler of guest OS

The guest OS scheduler should be capable of scheduling different processes.Currently the scheduler of Guest OS is not timing driven. It executes one-by-one instruction from each user process i.e. in round-robin way.

It should be further improved in a way such that it becomes time-driven. At a timer interrupt (every Δ seconds), control will reach the scheduler routine of the guest OS. The scheduler will save all CPU registers and PC of currently running process into a memory area. These CPU registers will form a part of the 'state' of the process that was interrupted by occurrence of the timer interrupt.

The simulator would then record the state of the interrupted user process in the PCB of that process and would now select the next process to be run. Its state would be available in its PCB. The scheduler does the following with it:

- It would load the CPU register contents as recorded in the CPU state, into the corresponding CPU registers, by using an X86 instruction.
- It would take the address stored in the PCB of the process that will be scheduled and will start its execution from its stored PC value.

Details of processes in guest OS

Multiple user processes(executables of C programs) can be run on the top of guest OS. For guest OS ,each such executable is a "process ". When a user runs the Guest OS system ,to run the user program under guest OS, he has to enter the path of executable when prompted. The creation of different processes one per user program is done in guestos.c file in main() function by calling ld_load_prog_from_ctxconfig(ctxconfig); Here ctxconfig has the details of user programs to be created. ld_load_prog_from_ctxconfig function is present in guestos/src/libm2skernel/loader.c . The context is created for each user process. So a list is formed of all contexts. The context of a process consists of:

```
 struct ctx_t {

/* Context properties */
int status;
int pid;  /* Context id */
int mid;  /* Memory id - the same for contexts sharing memory map */
struct ctx_t *parent;
int exit_signal;  /* Signal to send parent when finished */
int exit_code;  /* For zombie processes */

uint32_t initial_stack;  /* Value of esp when context is cloned */
      /* For segmented memory access in glibc */
uint32_t glibc_segment_base;
uint32_t glibc_segment_limit;
.
.
.
.
    /* Substructures */
struct loader_t *loader;
struct mem_t *mem;  /* Virtual memory image */
struct fdt_t *fdt;  /* File descriptor table */
struct regs_t *regs;  /* Logical register file */
struct signal_masks_t *signal_masks;
struct signal_handlers_t *signal_handlers;
};
```

After the processes are created and loaded in guest OS memory, the processes are executed in a way such that 1 instruction from each process. This is done by following code from function ke_run() present in guestos/src/libm2skernel/m2skernel.c

```
  /* Run an instruction from every running process */
for (ctx = ke->running_list_head; ctx; ctx = ctx->running_next)
ctx_execute_inst(ctx);
```

This way round-robin policy is implemented based on single instruction basis.

Chapter 5

GuestOS tutorial

5.1 Working with GuestOS

5.1.1 Downloading the GuestOS

Download GuestOS archive from `http://www.cse.iitb.ac.in/~abhaykadam/guestos`.

5.1.2 Compiling the GuestOS

Extract the given GuestOS archive at a directory, say guestos, where you want to keep the GuestOS. Follow the steps:

- Open a terminal
- create a directory named build in the same directory where guestos resides.
- cd into the build directory.
- Run

  ```
  ../guestos/configure && make
  ```

5.1.3 Running user programs under Guest OS

Follow the steps:

- cd build's root dicrectory
- Run

  ```
  ./src/guestos
  ```

  One point to note here is that your boot configuration file named .config needs to reside in the current working directory. copy it from guestos/src/.config to the present working directory.

- The user will be asked to enter the full path of executable to run

5.2 Boot Sequence

GuestOS' boot code installs system-call handlers, signal handlers and initializes the parameters like virtual disk's number of heads, tracks and sectors or instruction count which will be use to execute those many instructions in one slot alloted to the process. These parameters have to be defined in .config file in current working directory (the directory from which you are invoking the GuestOS).

Boot code is defined in function boot() in src/guestos.c file. After installing system calls and signals, boot() calls set_defaults() function, which reads values for parameters related to scheduling or file management from .config file and initializes them.

5.3 Process Context

Process context stores all the information related to the process, like process's ID, its parent ID, its status, link to the next process's context structure, etc. In GuestOS, the process context is a structure named ctx_t defined in src/libm2skernel/m2skernel.h header file.

The current process' context can be referred by using isa_ctx pointer variable of type ctx_t. The current register set of machine can be referred by using isa_regs pointer variable of type regs_t.

5.4 System Calls

System call functionality in GuestOS is divided into two parts: first part that contains system calls implemented by host operating system (i.e., the system calls numbered from 0 to 325). The second part contains system calls implemented by GuestOS. GuestOS' system call range starts from 400. The mapping from system call name to its number is declared in src/libm2skernel/syscall.dat file. The range of guest OS handled system calls can be easily increased to a desired number by adding their entries in files syscall.c and syscall.dat files.

Adding a new hello() system call to GuestOS:

- Add DEFSYSCALL(hello, 400+n) to the syscall.dat file, where n represents current number of GuestOS system calls.
- Write the code for hello() system call in src/libm2skernel/syscall.c file.

```
int hello(uint32_t name) {
  char person[20];

  int length = mem_read_string(isa_mem, name, 20, person);
  if (length >= 20)
    return -1;

  printf("Hello, %s\n", person);
```

```
  return 0;
}
```

mem_read_string() is a function defined in src/libm2skernel/memory.c and it reads the userland string to the GuestOS address space. We require to pass four arguments to the function. isa_mem is the current page table address (i.e. page table of the process which invoked the system call), name is the address of the string in user space, 20 is input buffer length, and person is the local character array to store the string in. mem_write_string() writes the string to the user buffer. For more information on memory user to kernel read/write functions refer section §5.5.

- Now add following code snippet in handle_guest_syscalls():

```
case syscall_code_hello:
{
  retval = hello(isa_regs->ebx);
  break;
}
```

- Hurray! you just wrote your first system call for GuestOS. you can invoke this through your test program as syscall(n,"name"), where n is the system call number you entered in syscall.dat file.

5.5 Memory Access Methods

All the memory related functions are defined in src/libm2skernel/memory.c file. As you saw in section §5.4, mem_read_string() and mem_write_string() are the functions to read from and and write to the user space string, respectively. These functions that's mem_read_string() and mem_write_string() calls the generic function mem_access(). mem_access() is responsible for accessing user pages for either reading or writing or executing. These operation mode is specified as a parameter for mem_access(). This parameter's value may either be one of the following: mem_access_read, mem_access_write, mem_access_exec.

To read or write user space buffers mem_read(), mem_write() macros are used. These macros are ultimatley expanded to call the mem_access() function with either mem_access_read or mem_access_write parameter in it.

mem_access() is declared as:

```
void mem_access(struct mem_t *mem, uint32_t addr, int size, void *buf,
  enum mem_access_enum access)
```

mem points to the memory page table of current process. addr is the address of the buffer in the user space whose contents are to be accessed. buf is the buffer in the

kernel space where the accessed contents are written or read from depending on whether mem_access() is called to read or write the user space contents. size specifies size of buf. access parameter specifies the mode of mem_access(), whether its being used for read or write.

5.6 Scheduling

GuestOS executes specified number of instructions of each process in round-robin manner. The default instruction-slice (i.e. the number of instructions that can be executed in one slot) is initialize at booting time and the istr_slice value is read from .config file. When the new process' context is created, it's instr_slice member is initialized to the default value set at the booting time. ke_run() defined in src/libm2skernel/m2skernel.c file is responsible for executing the processes.