Final Exam Notes

David Robinson

File Systems and Navigation

What makes the unix file system "hierarchical"?

It organizes files and directories, which are just special files, in a tree-like structure, where there is a root directory and each directory can contain other directories and files.

What is the difference between absolute vs. relative paths?

Absolute paths start from the root directory while relative paths start from the working directory.

How are parent directories referenced in the file system?

Parent directories are referenced with `..` in the filepath.

What is the working directory and how do you display it?

The working directory for a process is the directory where the process is operating and can be displayed with the `pwd` command.

What is the unix standard command to rename a file?

`mv old-filename new-filename`

What is tab-completion?

A feature that automatically completes commands and filepaths if there is only one option.

What unix standard will show you the text of a file?

`cat filename`

How do you change the working directory?

`cd path` in bash and `chdir()` as a system call, and can be used with relative or absolute file paths.

What is the unix command to delete a file?

`rm filename`

How does the implementation of deleting a file work? Does it remove the file's contents from the storage medium?

The file's metadata, including its directory entry in the parent directory, is removed, but the data remains on disk until it is overwritten by new data.

Version Control

What git command copies commits from the local repository to the remote repository?

`git push`

What git command copies commits from the remote repository to the local repository?

`git pull`

What git command stages a new file?

`git add filename`

What git command creates a log of the change to a staged file to the local repository?

`git commit`

Processes, Advanced Processes

How do you redirect standard in or out of bash command to a file?

Use `<` to redirect standard in and `>` to redirect standard out.

How do you redirect standard out from one command to another command's standard in?

Use the pipe `|` command between the two comamnds.

Editor

How do you edit files in vim?

`vim filename`

How do you quit the editor in vim?

`:q`

Processes, Pipe, Syscalls

How are processes created with unix system calls?

The `fork()` command creates a new child process by duplicating the calling process.

```
pid_t pid = fork();

switch (pid) {
case -1:
    perror("fork");
    exit(EXIT_FAILURE);
case 0:
    // Child process
    _exit(EXIT_SUCCESS);
default:
    // Parent process
    break;
}
```

How are programs executed with unix system calls?

The current process is forked and the child process is replaced with the new program, using any of the exec system calls (exel, execv, execle, execve, execlp).

```
// cat filename
char *args[] = {"cat", "filename", NULL};
execvp("cat", args);
```

How are standard in, out, and err redirected with system calls?

`dup2` can be used to redirect the standard in, out, and err to a new file.

```
int fd = open("data.txt", O_RDWR | O_CREAT, 0644);
dup2(fd, STDOUT_FILENO);
dup2(fd, STDERR_FILENO);
dup2(fd, STDIN_FILENO);
```

How do pipes work and what do they do?

The pipe system call creates a temporary file in memory with a pair of connected file descriptors where one is used for writing data to the pipe and the other is used for reading data. In the pipe bash command, the stdout of the first process is redirected to the pipe's write end and the stdin of the second process is redirected to the pipe's read end.

Source Code vs. Executable

How are executables created from source code?

Starting with the source code (.c), it is preprocessed (.i), compiled into assembly language (.s), converted to machine language (.o), and linked with libaries and other object files to create a binary file that can be executed.

What unix utilities are involved in each phase?

1. `cpp` — C Preprocessor
2. `gcc` — C Compiler
3. `as` — Assembler
4. `ld` — Linker

SimpleIR

What instructions are available?

1. Local variables — variable declaration, variable assignment
2. Pointer operations — reference, dereference
3. Arithmetic operators — add, subtract, multiply, divide, modulus
4. Branching — labels, goto statements, if goto statements
5. Function calls and return statements

SimpleIR Grammar

```
grammar SimpleIR;

unit: function;

function: 'function' NAME localvars? params? statement* return;

localvars: 'localvars' NAME+;

params: 'params' NAME+;

return: 'return' operand=(NAME | NUM);

statement: assign | deref | ref | assignderef | operation | call | label |
    goto | ifgoto;

operation: NAME ':=' operand1=(NAME | NUM)
    operator=('+' | '-' | '*' | '/' | '%') operand2=(NAME | NUM);

assign: NAME ':=' operand=(NAME | NUM);

deref: NAME ':=' '*' NAME;

ref: NAME ':=' '&' NAME;

assignderef: '*' NAME ':=' operand=(NAME | NUM);

call: NAME ':=' 'call' NAME NAME*;

label: NAME ':';

goto: 'goto' NAME;

ifgoto: 'if' operand1=(NAME | NUM)
    operator=('=' | '!=' | '<' | '<=' | '>' | '>=')
    operand2=(NAME | NUM) 'goto' labelname=NAME;

NAME: [a-zA-Z_] ([a-zA-Z_] | [0-9])* ;
NUM: [0-9]+ ;

PLUS: '+' ;
MINUS: '-' ;
STAR: '*' ;
SLASH: '/' ;
PERCENT: '%' ;

EQ: '=' ;
NEQ: '!=' ;
LT: '<' ;
LTE: '<=' ;
GT: '>' ;
GTE: '>=' ;

WS:   [ \t\r\n]+ -> skip ;

COMMENT : '#' ~[\r\n]* -> skip ;
```

Sample SimpleIR Programs

main.ir

```
function main
localvars a b x y t1 t2 t3
params a b
x := 8
y := 9
t1 := &x
t2 := *t1
*t1 := 11
t3 := x
return x
```

arithmetic.ir

```
function main
localvars argc argv x
params argc argv
x := 1 + 2  # 3
x := x * 6  # 18
x := x - 1  # 17
x := x % 10 # 7
x := x / 3  # 2
return x    # 2
```

branching.ir

```
function main
localvars a b x y t1 t2
params a b
x := 9
y := 4
if x < 10 goto true1  # true
t1 := x + y
goto end1
true1:
t1 := x - y  # 5
end1:
if x > 10 goto true2  # false
t2 := x * y  # 36
goto end2
true2:
t2 := x / y
end2:
return x  # 9
```

main.s

```
    .file "main.ir"
    .section .note.GNU-stack,"",@progbits
    .text
    .globl main
    .type main, @function
main:
    # prologue, update stack pointer
    pushq %rbp # save old base ponter
    movq %rsp, %rbp # set new base pointer
    push %rbx # %rbx is callee-saved
    # allocate stack space for locals
    sub $56, %rsp
    # move register parameter a to local variable
    mov %rdi, -8(%rbp)
    # move register parameter b to local variable
    mov %rsi, -16(%rbp)
    # assign 8 to x
    mov $8, %rax
    mov %rax, -24(%rbp)
    # assign 9 to y
    mov $9, %rax
    mov %rax, -32(%rbp)
    # ref x to t1
    mov %rbp, %rax
    add $-24, %rax
    mov %rax, -40(%rbp)
    # deref t1 to t2
    mov -40(%rbp), %rax
    mov (%rax), %rbx
    mov %rbx, -48(%rbp)
    # assignderef 11 to t1
    mov -40(%rbp), %rax
    mov $11, %rbx
    mov %rbx, (%rax)
    # assign x to t3
    mov -24(%rbp), %rax
    mov %rax, -56(%rbp)
    # set return value
    mov -24(%rbp), %rax
    # epilogue
    pop %rbx # restore rbx for the caller
    mov %rbp, %rsp # restore old stack pointer
    pop %rbp # restore old base pointer
    ret
```

arithmetic.s

```
    .file "arithmetic.ir"
    .section .note.GNU-stack,"",@progbits
    .text
    .globl main
    .type main, @function
main:
    # prologue, update stack pointer
    pushq %rbp # save old base ponter
    movq %rsp, %rbp # set new base pointer
    push %rbx # %rbx is callee-saved
    # allocate stack space for locals
    sub $24, %rsp
    # move register parameter argc to local variable
    mov %rdi, -8(%rbp)
    # move register parameter argv to local variable
    mov %rsi, -16(%rbp)
    # operation 1 2 to x
    mov $1, %rax
    mov $2, %rbx
    add %rbx, %rax
    mov %rax, -24(%rbp)
    # operation x 6 to x
    mov -24(%rbp), %rax
    mov $6, %rbx
    imul %rbx, %rax
    mov %rax, -24(%rbp)
    # operation x 1 to x
    mov -24(%rbp), %rax
    mov $1, %rbx
    sub %rbx, %rax
    mov %rax, -24(%rbp)
    # operation x 10 to x
    mov -24(%rbp), %rax
    mov $10, %rbx
    cdq
    idiv %rbx
    mov %rdx, %rax
    mov %rax, -24(%rbp)
    # operation x 3 to x
    mov -24(%rbp), %rax
    mov $3, %rbx
    cdq
    idiv %rbx
    mov %rax, -24(%rbp)
    # set return value
    mov -24(%rbp), %rax
    # epilogue
    pop %rbx # restore rbx for the caller
    mov %rbp, %rsp # restore old stack pointer
    pop %rbp # restore old base pointer
    ret
```

branching.s

```
main: (skipped metadata and prologue)
    # allocate stack space for locals
    sub $56, %rsp
    # move register parameter a to local variable
    mov %rdi, -8(%rbp)
    # move register parameter b to local variable
    mov %rsi, -16(%rbp)
    # assign 9 to x
    mov $9, %rax
    mov %rax, -24(%rbp)
    # assign 4 to y
    mov $4, %rax
    mov %rax, -32(%rbp)
    # ifgoto x 10 to true1
    mov -24(%rbp), %rax
    mov $10, %rbx
    cmp %rbx, %rax
    jl true1
    # operation x y to t1
    mov -24(%rbp), %rax
    mov -32(%rbp), %rbx
    add %rbx, %rax
    mov %rax, -40(%rbp)
    # goto
    jmp end1
true1:
    # operation x y to t1
    mov -24(%rbp), %rax
    mov -32(%rbp), %rbx
    sub %rbx, %rax
    mov %rax, -40(%rbp)
end1:
    # ifgoto x 10 to true2
    mov -24(%rbp), %rax
    mov $10, %rbx
    cmp %rbx, %rax
    jg true2
    # operation x y to t2
    mov -24(%rbp), %rax
    mov -32(%rbp), %rbx
    imul %rbx, %rax
    mov %rax, -48(%rbp)
    # goto
    jmp end2
true2:
    # operation x y to t2
    mov -24(%rbp), %rax
    mov -32(%rbp), %rbx
    cdq
    idiv %rbx
    mov %rax, -48(%rbp)
end2:
    # set return value
    mov -24(%rbp), %rax
    # epilogue
    pop %rbx # restore rbx for the caller
    mov %rbp, %rsp # restore old stack pointer
    pop %rbp # restore old base pointer
    ret
```

Function Implementation

Stack frames, its contents

The stack frame stores any remaining parameters after the first six, return address, old base pointer, and local variables, in that order from high address to low address.

System V AMD64 ABI

The Application Binary Interface that defines how software components, such as functions or libraries, interact at the binary level on the AMD64/x86-64 architecture, such as passing parameters and setting up the stack with prologue and epilogue.

Parameter passing

The first six parameters are stored in the registers: RDI, RSI, RDX, RCX, R8, and R9, and the rest are pushed onto the stack.

Prologue and epilogue

The prologue saves the base pointer and sets a new one, allocates space for local variables, and pushes any callee-saved registers onto the stack. The epilogue restores the stack pointer, base pointer, and any registers that the function modified, as well as returning to the caller's instruction pointer.

call and ret behavior

The `call` instruction saves the return address and branches to the function. The `ret` instruction pops the return address and branches to it.

Comparison of SimpleIR functions to assembly "functions"

An assembly function call is just a branch while a SimpleIR function call includes passing all the parameters to the registers and stack, saves the return address, branches to the function, and executes the function prologue.

Variables

Local variable layout in the stack frame

Local variables are stored below the base pointer and the stack pointer (rsp) decreases as more data is pushed onto the stack. In the SimpleIR language the pointer, decreases by 8 so the first variable is stored at `-8(%rbp)`, second at `-16(%rbp)`, etc.

Assignment, both immediate values and from other variables

Set the value of the rax register to the immediate value or value of other variable, then assign the value of rax to the new variable.

```
mov $1, %rax
mov %rax, -16(%rbp)

mov -16(%rbp), %rax
mov %rax, -24(%rbp)
```

The compiler's symbol table

Maps each variable to its offset from the base pointer in the stack

Variable	Offset
x	-8
a	-16
b	-24

Assembly instruction support for accessing variables in the stack frame

`offset(%rbp)`

Arithmetic

What x86-64 instructions are used for each arithmetic operator in SimpleIR?

1. Addition: `add`
2. Subtraction: `sub`
3. Multiplication: `imul`
4. Dividion and Modulus: `cdq` and `idiv`

SimpleIR arithmetic commands vs. x86-64 assembly (number of arguments, behavior)

Store the two operands to %rax and %rbx, perform the operation, and store the result to the x variable in the stack. Also, SimpleIR command takes in the resulting variable, while the x86-64 command stores the result in one of the registers.

1. Addition: `x := a + b` vs. `add %rbx %rax` (sum stored in %rax)
2. Subtraction: `x := a - b` vs. `sub %rbx %rax` (difference stored in %rax)
3. Multiplication: `x := a * b` vs. `sub %rbx %rax` (product stored in %rax)
4. Division and Modulus: `x := a / b` or `x := a % b` vs. `cdq`, `idiv %rbx` (quotient stored in %rax and remainder stored in %rdx)

Branching

Unconditional branching in SimpleIR vs. x86-64 assembly

`goto label` vs. `jmp label`

Conditional branching in SimpleIR and x86-64 assembly

SimpleIR

```
    if x <= 10 goto falselabel
    x := 11
falselabel:
    x := x + 1
    return x
```

x86-64 Assembly

```
    # ifgoto x <= 10 to falselabel
    mov -8(%rbp), %rax
    mov $10, %rbx
    cmp %rbx, %rax
    jle falselabel
    # assign 11 to x
    mov $11, %rax
    mov %rax, -8(%rbp)
falselabel:
    # operation x + 1 to x
    mov -8(%rbp), %rax
    mov $1, %rbx
    add %rbx, %rax
    mov %rax, -8(%rbp)
```

How if, while, and for can be implemented in SimpleIR

1. If statement: Set a label at the end of the statement block. Add an `ifgoto` statement at the beginning of the statement block that jumps to the label if the condition is false.
2. While loop: Set labels at the beginning and end of the statement block. Right before the start label, add an `ifgoto` statement that jumps to the end label if the condition is false. Right before the end label, add an `ifgoto` statement that jumps to the beginning label if the condition is true.
3. For loop: Do the same as while loop but add the initialization code before the beginning ifgoto statement and add the update code before the end ifgoto statement.

Jump instructions in x86-64 assembly

je, jne, jl, jle, jg, jge

Pointers

SimpleIR's three pointer operations: reference, dereference, and dereference assignment

1. Reference: `t1 := &x`
2. Dereference: `t2 := *t1`
3. Dereference assignment: `*t1 := 11`

x86-64 mov instructions for performing register indirect stores and loads

1. Register indirect store: `mov (%rax), %rbx` (stores the value in rbx into the memory address stored in rax)
2. Register indirect load: `mov %rax, (%rbx)` (loads the value at the memory address stored in rbx into rax)

Implementing SimpleIR's pointer operations in x86-64 assembly

If offset of x is -24, offset of t1 is -40, and offset of t2 is -48.

1. Reference:

```
mov %rbp, %rax
add $-24, %rax
mov %rax, -40(%rbp)
```

2. Dereference:

```
mov -40(%rbp), %rax
mov (%rax), %rbx
mov %rbx, -48(%rbp)
```

3. Dreference assignment:

```
mov -40(%rbp), %rax
mov $11, %rbx
mov %rbx, (%rax)
```