CMSC 313
Introduction to Computer Systems
Lecture 27
Procedure Implementation & Testing Programs

Alan Sussman
als@cs.umd.edu

Administrivia

- Project 6 due tomorrow
  - questions?
- Project grade weights
  - total of 35% of your grade – each 6%, except Project 3 is 5%
- Practice final exam posted this afternoon, with answers soon
- Final exam is on Wed., Dec. 16, 1:30-3:30PM
  - in CSIC 3117
- I'll have Quiz 5, graded, tomorrow in my office
- Please do course evaluation, at www.CourseEvalUM.umd.edu

Section 3.7, Bryant and O'Hallaron

PROCEDURE IMPLEMENTATION

Parameter example

```
void f(int);

int main() {
  f(72);
  printf("\n");
  return 0;
}

void f(int i) {
  printf("%d", i);
}
```

```
main: irmovl $0x1000, %esp  # init stack ptr
      irmovl $72, %eax      # load arg to reg
      pushl  %eax           # push arg on stack
      call   f
      irmovl $10, %eax      # printf("\n");
      wrch   %eax
      halt                  # return 0;
f:    pushl  %ebp           # save old frame ptr
      rrmovl %esp, %ebp     # set new frame ptr
      mrmovl 8(%ebp), %eax  # load i param
      wrint  %eax           # print i
      rrmovl %ebp, %esp     # reset stack ptr
      popl   %ebp           # restore frame ptr
      ret
```

Recursive parameter example

```
void f(int);

int main() {
  f(5);
  printf("\n");
  return 0;
}

void f(int i) {
  if (i > 0)
    f(i-1);
  printf("%d", i);
}
```

```
main:  irmovl $0x1000, %esp  # init stack ptr
       irmovl $5, %eax       # load arg to reg
       pushl  %eax           # push arg on stack
       call   f
       irmovl $10, %eax      # printf("\n");
       wrch   %eax
       halt                  # return 0;
f:     pushl  %ebp           # save old frame ptr
       rrmovl %esp, %ebp     # set new frame ptr
       mrmovl 8(%ebp), %eax  # load i param
       irmovl $0, %ecx       # load 0 to reg
       addl   %ecx, %eax     # test value of i
       jle    EndIf          # if i > 0, rec. call
       irmovl $-1, %ecx
       addl   %eax, %ecx     # create i-1 arg
       pushl  %ecx           # push arg on stack
       call   f
EndIf: mrmovl 8(%ebp), %eax  # load i param again
       wrint  %eax           # print i
       rrmovl %ebp, %esp     # reset stack ptr
       popl   %ebp           # restore frame ptr
       ret
```


Register usage conventions

- Notice in the last example we had to reload the value of the parameter **`i`** after the recursive function call, because recursive calls to **`f`** would destroy the value in **`%eax`**
- A convention has developed as to which registers can be used by a procedure without destroying data from the caller
- When P calls Q...
  - Q can do anything it wants with the *caller save* registers; P is required to store these before the call if there's important data there
  - Q must save the values of the *callee save* registers, and restore them before returning (if Q uses these registers); P can use these to maintain data across function calls
- For reference (i.e., don't memorize this), and by convention, **`%eax`**, **`%edx`**, and **`%ecx`** are caller save, while **`%ebx`**, **`%esi`**, and **`%edi`** are callee save

Using callee save registers

```
f:     pushl  %ebp
       rrmovl %esp, %ebp

       mrmovl 8(%ebp), %eax
       irmovl $0, %ecx
       addl   %ecx, %eax
       jle    EndIf
       irmovl $-1, %ecx
       addl   %eax, %ecx
       pushl  %ecx
       call   f

EndIf: mrmovl 8(%ebp), %eax
       wrint  %eax
       rrmovl %ebp, %esp
       popl   %ebp
       ret
```

```
f:     pushl  %ebp
       rrmovl %esp, %ebp
       pushl  %ebx           # store callee save
       mrmovl 8(%ebp), %ebx  # holds value of i
       irmovl $0, %ecx
       addl   %ecx, %ebx
       jle    EndIf
       irmovl $-1, %ecx
       addl   %ebx, %ecx
       pushl  %ecx
       call   f
       popl   %ecx           # pop arg off stack
EndIf: wrint  %ebx           # no need for reload
       popl   %ebx           # restore val of reg
       rrmovl %ebp, %esp
       popl   %ebp
       ret
```


Returning values

- In Y86 (and x86), the convention is to store return values in **`%eax`**, and have the caller read this value after the call returns
- This works well enough for integers, but what about something like structs?
  - If the struct has ≤ 3 values, you could use all the caller save registers to pass the value back
  - The struct could be placed in the stack and accessed immediately after the function call

Example of returning values

```
int f(int);

int main() {
  static int n;
  n = f(5);
  printf("%d", n);
  printf("\n");
  return 0;
}

int f(int i) {
  return (i+1) * 3;
}
```

```
main:  irmovl $0x1000, %esp  # init stack ptr
       irmovl $5, %eax       # load arg to reg
       pushl  %eax           # push arg on stack
       call   f
       rmmovl %eax, n        # store return value
       mrmovl n, %eax        # load value of n
       wrint  %eax
       irmovl $10, %eax      # printf("\n");
       wrch   %eax
       halt                  # return 0;

f:     pushl  %ebp           # save old frame ptr
       rrmovl %esp, %ebp     # set new frame ptr
       mrmovl 8(%ebp), %eax  # load param i
       irmovl $1, %ecx       # m + 1
       addl   %ecx, %eax     # add 1
       irmovl $3, %ecx       # for * 3
       multl  %ecx, %eax     # multiply by 3
       rrmovl %ebp, %esp     # reset stack ptr
       popl   %ebp           # restore frame ptr
       ret

       .align 4
n:     .long 0
```


Not in the text

TESTING PROGRAMS

Testing

- Why test software?
  - find bugs in the implementation
  - find mistakes in the specification
- Testing is a major part of software development
  - around 50% of time in many systems is spent on testing
  - for life-critical software, it can be 3-5 times as much time as actual development
- The testing process
  - execute a program with the intent of finding *errors*
  - run special code (or inputs) called *test cases*
  - the goal is that each new test should increase the likelihood of finding a yet-undiscovered bug

What do we try to test?

- Normal cases
  - does the program work on valid inputs or parameters?
    - does it not crash?
    - does it perform as expected?
  - these are generally the easiest type of tests to write
- Error cases
  - does the program work on invalid inputs or parameters?
    - does it crash?
    - what happens **after** an error (can we continue)?
  - often most bugs lie in this part of the code, since it has many infrequently-executed cases
- Environmental errors
  - these situations are often very hard to test
  - examples: computer out of memory, disk drive inaccessible

Tests are an integral part of the software

- Write tests as you write software
  - don't wait until code is done to write tests
  - sometimes test code is written **before** writing the software itself
  - test each function/method as you write it
- Test code can be large
  - a sample project: Dyninst is 133,000 lines, with 16,000 lines of testing code
- When the software is "done", tests continue as part of it
  - it's necessary to be able to re-test when the code changes
  - it's necessary to validate the code on new platforms (e.g., 64-bit systems), new versions of the operating system or compiler, and sometimes even individual systems

White box testing

- Examine the statements of a program to be tested in order to create test cases
  - look for places errors can occur
  - write test cases designed to have the program run all possible combinations
  - write test cases designed to hit boundary conditions
- Try to get the maximum **coverage** of a test and test suite
  - coverage can be expressed or measured in terms of:
    - lines of a program (*line* or *statement coverage*)
    - control points in a program (*branch coverage*)
    - sequences of statements (*path coverage*)
  - try to write the minimum set of tests to achieve an acceptable coverage level. 100% coverage is rarely possible

White-box testing- how to test different types of statements

- If-then-else
  - are all cases executed by at least some test?
- Switch
  - are all branches executed?
- Loops
  - do some inputs run the loop:
    - not at all
    - only one time
    - two times
    - a large number of times
  - test all possible termination conditions
- Assignment statements
  - use a range of possible values
    - for `int`s: positive, negative, 0, 1, and any special values

Black box testing

- Create tests for a program to be tested based on the program's specifications, without looking at its code
  - does it perform as expected
  - don't examine how it does the job, just does it do it?
- Try to look for:
  - incorrect or missing functions or behavior
  - interface errors
  - errors in data structures or external database access
    - after the call is the file or database correct?
  - performance errors
    - timing specifications can be as critical as correctness specifications
  - initialization and termination errors
    - is all allocated memory freed?
    - is all memory used initialized?

Notes on testing

- Often tests may be a hybrid of white and black box
- You can spend near infinite amounts of effort on testing
  - concentrate on tests that demonstrate classes of problems
- Approach testing with the mindset that whatever can go wrong will go wrong

Using lcov

- A statement coverage tool for Linux (derived from the Free Software Foundation's gcov tool)
- Steps:
  1. Compile the program with flags
     `-fprofile-arcs -ftest-coverage`
  2. Run the program on one or more test cases
  3. **`lcov -c -d . -o coverage.out`**
  4. **`genhtml coverage.out`**
  5. Creates a file **`index.html`** in the current directory, which can be viewed with a web browser