CS Basics
6) Procedures

Fall Term 2023-24

E.Benoist, C.Fuhrer, Ch. Grothoff, L. Ith, P.Mainini | BFH-TI

Procedures in Assembly

- **Procedures**
- **Program Data**
- **Local Labels**
- **Libraries**
  Example: hexdump2
- **Command Line Arguments**
  Example: showargs
- **Entering the World of C**
  Examples of Calling C Functions
- **Conclusion**

Procedures

The Need for Procedures

Issues with large monolithic programs

- Very hard to keep overview
- Impossible to test properly
- Tasks have to be done sequentially

Procedures

- Used for finer granularity in programming
- Can be called once or multiple times
- Can be reused, also in other programs

Examples for procedures

- Read a string from `stdin`
- Transform a string into a number
- Compute logarithms
- ...

Procedures

What is a procedure?

- Just another part of code
- Intended to be called from somewhere else
- Returns to the calling code afterwards (similar to `syscall`)

What is the difference to jumps?

- Jumps do not provide a direct returning mechanism
- Jumps should remain inside a procedure

Calling a Procedure I

The `call` instruction is used for calling procedures:

```
call read
```

Example procedure: `read`
A procedure which reads from `stdin` (and "returns" the number of bytes read in `rax`)

```
read:
    mov rax, 0              ; sys_read
    mov rdi, 0              ; file descriptor: stdin
    mov rsi, InBuf          ; destination buffer
    mov rdx, InBufLen       ; maximum # of bytes to read
    syscall
    ret
```

Calling a Procedure II

First, `call` pushes the return address on the stack

- The return address is the next value of the instruction pointer

Then transfers execution to the address/label given

The procedure finally returns using the instruction `ret`

- Pops the return address from the stack to the instruction pointer
- Continues execution just after `call`

Similar to `syscall`

- But: No switch between kernel- and user space
- Destination address is known
- As opposed to `syscall`, where only syscall numbers are known

Calling a Procedure III

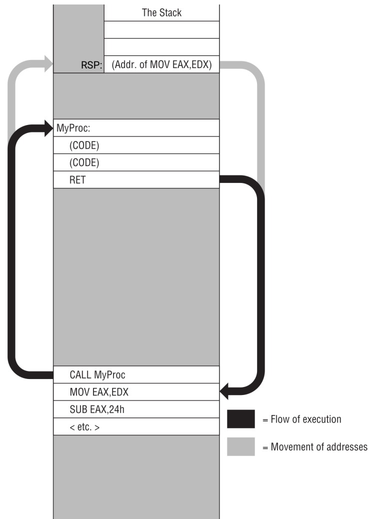

Figure 10-1: Calling a procedure and returning

Using `call` in Procedures

In a procedure, all instructions are allowed

- Including: calling a procedure

```
; Procedure: Resets all dumped characters in a line to 0
; Input: -
; Returns: -
clear_line:
    push rbx                                ; rbx is callee-saved, save it
    mov rbx, 15                             ; loop 16 times
.loop:
    mov rax, 0                              ; dump_char(0)
    call dump_char
    sub rbx, 1                              ; decrement loop counter
    jae .loop                               ; loop if rbx >= 0
    pop rbx                                 ; restore rbx
    ret
```

Recursion

- ***Definition of Recursive Procedures***
  - A procedure is *recursive* if it calls itself
  - It needs a *terminating condition* which stops recursive calls
- **Example: Exponential function**
  - Goal: Compute x^y
    - Test if $y = 0$; if yes: return 1 *(terminating condition)*
    - Otherwise, the result is $x \cdot z$ where $z = x^{y-1}$ *(recursion)*
- **Risk: Stack overflow**
  - Terminating condition is never met $\rightarrow$ endless recursion!
  - Every procedure call requires space on the stack (e.g. return address)
  - Sooner or later, the stack collides with other memory regions, leading to undefined behavior[1]
- **Further reading**
  - *Tail recursion* solves above issue. More about recursion in the algorithms and data structures module...

[1] On GNU/Linux, this typically results in a page fault.

Program Data

Types of Data

- **In general, procedures work on data**
  - Input data (parameters/arguments)
  - Return value
  - Internal, temporary data
- **We distinguish *global* and *local* data**
- **Global data**
  - Accessible from anywhere in the program
  - Is defined in `.data` or `.bss` sections
  - *CPU registers also have to be considered global*
- **Local data**
  - Typically stack and heap, but also registers...
- **Often, registers are enough**
  - Simple procedures
  - Parameters for syscalls

Saving Registers I

Registers are a scarce resource

- Limited number of registers, used by all parts of a program
- Normally, always in use

In general, registers must be saved when switching between procedures

- It is typically not known if/how a register is used by code in a different procedure

Solution: Push registers on the stack

- A register used by a procedure can be saved on the stack before modification
- When the procedure is done, it is restored from the stack

Note for `pusha` and `pushad` (not introduced)

- Not valid in 64-bit mode!
- Registers must be pushed individually

Saving Registers II

Registers modified in the procedure are normally saved at the beginning and restored at the end

```
procedure_x:
    push rbx
    push r12

    ...

    pop r12       # pop in reverse order!
    pop rbx
    ret
```

Calling Conventions (ABI)

When interacting with *foreign* code, register usage must be agreed upon!

- E.g. when using external libraries

→*Calling conventions*, part of the so-called *Application Binary Interface (ABI)*

- Depend on CPU architecture, operating system, sometimes programming language / compiler
- For x86-64, normally either *System V AMD64 ABI*[2] or *Microsoft x64 ABI*

Caller- vs. callee-saved

- *Caller-saved (volatile) registers:* Caller has to save register *before* `call`
- *Callee-saved (non-volatile) registers:* Procedure has to save register. For x86-64: RBP, RBX and R12–R15

[2] `https://raw.githubusercontent.com/wiki/hjl-tools/x86-psABI/x86-64-psABI-1.0.pdf`

Local Data

Only accessible to a particular procedure

- Data placed on the stack or the heap while a procedure is executed

May be passed as arguments by the caller by `push`ing *before* `call`ing the procedure

- Data cannot be `pop`ed directly: return address has been placed on top of the stack by `call`
- Items on the stack must be addressed individually using the stack pointer

Hack: Define Constant Data in `.text`

Possibility to store “local” data within the `.text` section

- Recall: Instructions are just data
- If defined after `ret`, it will not be executed
- Can be addressed using a label
- Advantage: Read-only, Data not separated from procedure definition
- *Disadvantage: Read-only,*[3] *bad style*

```
procedure_x:
    push rbx
    push r12

    ...
    mov rsi, MyStr
    ...
    ret

MyStr:   db "Hello local World!",10
```

[3] BTW: For read-only data, there is also section “`.rodata`”

Local Labels

Local Labels I

When programs get larger...

- More and more labels are required (for loops, jumps, ...)
- Labels must be unique

Solution: Local labels

- Names start with a period (.)
- Can be referenced only below the corresponding global label
- May be globally accessed by concatenation (i.e. "`clear_line.loop`" in the example below)

Example

```
clear_line:
    push rbx                        ; rbx is callee-saved, save it
    ...
.loop:
    mov rax, 0                      ; dump_char(0)
    ...
    jae .loop                       ; loop if rbx >= 0
```

Local Labels II

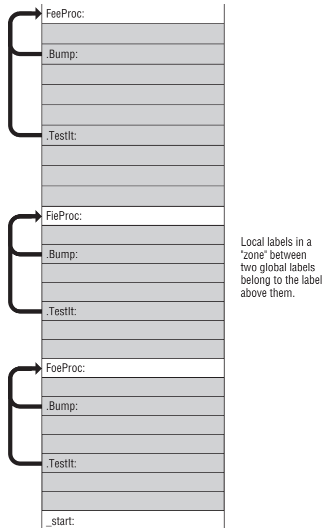

Figure 10-2: Local labels and the globals that own them

Short, Near and Far Jumps

If you obtain the error “Short jump is out of range”:

- Means that the jump destination is too far away
- Solution in general: change “`jmp`” to “`jmp near`”

There are actually three different types of jumps...

- *Short (default):* Within 127 bytes of the jump instruction
- *Near:* Inside the same code segment
- *Far:* Anywhere else (not relevant in flat memory models)

Example

```
jne read        ; Short, within 127 bytes in either direction
jne near read   ; Near, anywhere in the same code segment
```

Libraries

Libraries I

Issues with big, monolithic programs

- Difficult to maintain
- Error-prone
- Same code used in many different places
- Efficiency (e.g. disk space)

What is a library?

- Collection of useful procedures and data
- Just another piece of code
- Assembled and maintained independently
- Written and tested once
- Reusable in many programs

Libraries II

Creating libraries in assembly

- Separate file
- Contains the definition of the procedure(s)
- Procedures to be used from the outside need to be declared as `global`

Using procedures from libraries

- Code using library procedures needs to declare them as `extern`
- Library procedures can be used like procedures in the same file
- Constants and variables can also be defined `extern`

Linking program and libraries

- External references stay unresolved during assembly
- The linker (`ld`) stitches all the parts together

Libraries III

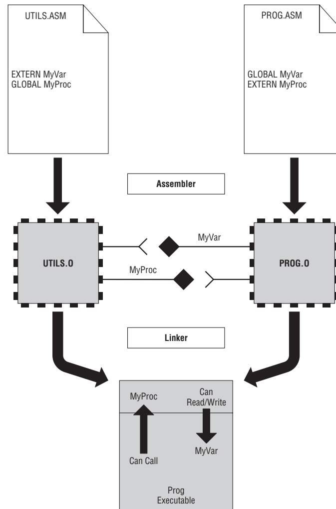

Figure 10-3: Connecting globals and externals

Libraries IV

Example: Creating a library

- Regular `.asm`-file containing the usual sections
- All public procedures, variables and constants must be declared as `global`
- *All labels (including “`_start`”) may only be defined once globally*

```
SECTION .data

global xy

    ...

SECTION .text

global clear_line, dump_char, print_line

clear_line:
    ...

dump_char:
    ...

print_line:
    ...
```

Libraries V

Example: Calling a library procedure

- Define as external, use as it would be in the same file
- Registers etc. must be set up as required by the procedure

```
SECTION .text

extern clear_line, dump_char, print_line

global _start

_start:
    ...
    call clear_line
    ...
```

Libraries VI

Assembling with libraries

- Until now: `makefile` assembles a single `.asm` file

```
hexdump1: hexdump1.o
    ld -o hexdump1 hexdump1.o
hexdump1.o: hexdump1.asm
    nasm -f elf64 -g -F dwarf hexdump1.asm
```

- Example: `makefile` for program + library

```
hexdump2: hexdump2.o textlib.o
    ld -o hexdump2 hexdump2.o textlib.o
hexdump2.o: hexdump2.asm
    nasm -f elf64 -g -F dwarf hexdump2.asm
textlib.o: textlib.asm
    nasm -f elf64 -g -F dwarf textlib.asm
```

`hexdump2.asm`

See `examples/` directory.

Command Line Arguments

Command Line Arguments

Running a program with arguments in the shell

- Arguments are separated with spaces

```
$ ./myprogram arg1 arg2 ... argN
```

Passing arguments in gdb

- Either `run` with arguments

```
(gdb) run arg1 arg2 ... argN
```

- Or use “`set args`”

```
(gdb) set args arg1 arg2 ... argN
(gdb) run
```

GNU/Linux Stack Initialization I

- **Stack is initialized by GNU/Linux at program start with**
  - Count of arguments
  - Addresses of arguments
  - Addresses of environment variables
  - Additional OS data (not further detailed here)
  - Effective argument and environment data
- **RSP points to the top of stack (count of arguments)**
- **Count of arguments**
  - A number ≥ 1 (invocation text is counted!)
- **Addresses of arguments**
  - Point to string data of arguments passed to the executable
  - First is *invocation text*: path and name of the executable invoked
  - Terminated by a 64-bit "null pointer" (8 bytes of 0)
- **Addresses of environment variables**
  - Point to string data containing the environment variables
  - Each also terminated by 64-bit null pointer

GNU/Linux Stack Initialization II

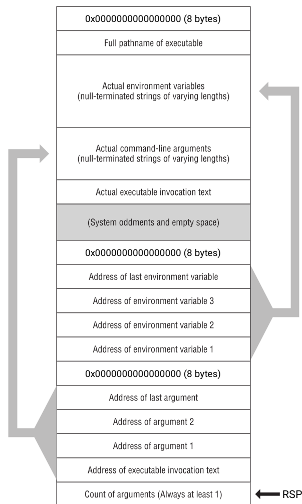

Figure 11-4: Linux stack at program startup

`showargs.asm`

See `examples/` directory.

Entering the World of C

Entering the World of C

```
#include <stdio.h>

int main(void) {
  puts("Eat at Joe's!"); // or rather: Hello world!
}
```

Unix is written in C

- The C language was conceived to implement Unix
- Large parts of the Linux kernel are written in C
- There are many (system) libraries written in C

Functions from C libraries can be called from assembly

- Passing of parameters must be understood
- Libraries must be linked to the assembler program

Objectives for now:

- Basic understanding of C calling conventions
- Access to a large library of useful C functions

Using GCC

Compiling using GCC

- Compiling and linking is done in one step by calling `gcc`

```
$ gcc -std=c17 -Wall -Wextra -Wpedantic -O3 hello.c -o hello
```

- The linking step may also be done separately, however, this requires correct `$OPTIONS` to link against libc, which is non-trivial

```
$ gcc -std=c17 -Wall -Wextra -Wpedantic -O3 -c hello.c
$ ld -o hello hello.o $OPTIONS
```

`gcc` performs compilation in multiple *stages*

- `.c` → `.c`: preprocessing with `cpp`
- `.c` → `.s`: assembly code generation
- `.s` → `.o`: assembling using `gas` (GNU Assembler)
- `.o` → executable: linking with `ld`

Note: All intermediary output can be inspected with “`--save-temps`”.

How gcc Builds Linux Executables

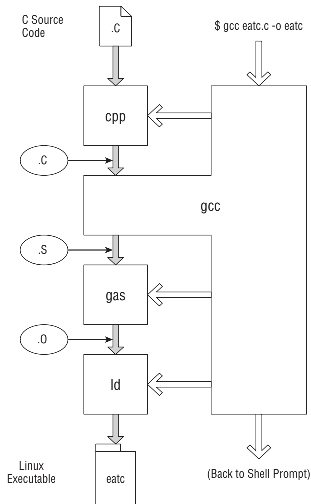

Figure 12-1: How gcc builds Linux executables

Using C Libraries

Link against C standard library[4]

- To access many useful functions, e.g. for printing strings (`puts`, `printf`), reading chars (`getchar`), accessing time (`time`), ...

Other or custom C libraries

- Requires the object file (`.o`) of the library
- Rule: Only one “`main`” procedure (assembly) or function (C) per executable

Your program is now a function called by the startup code of the C standard library

- `_start` is provided by the C standard library
- Your code must provide a `main` procedure (case sensitive)
- Do not exit the program with the sys_exit `syscall`, simply return from `main` using `ret`

Use `gcc` for linking!

[4] On GNU/Linux in general `glibc`

Structure of a Hybrid C/Assembly Program

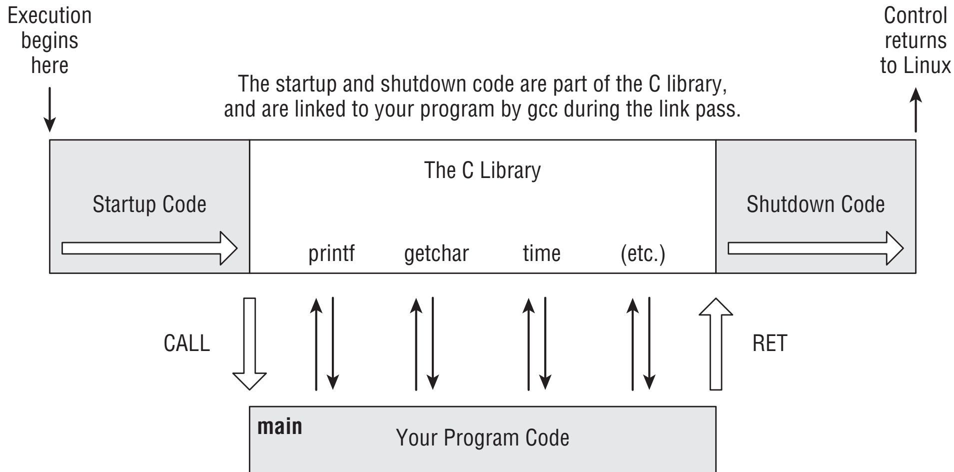

Figure 12-2: Structure of a hybrid C-assembly program

GNU/Linux x86 Calling Conventions *(Obsolete)*

When calling C functions with the `call` instruction, rules must be followed

- A procedure must preserve the values of EBX, ESP, EBP, ESI, and EDI. All other registers may be altered: registers have to be saved prior calling a function!
- Return value is stored in EAX (32-bit results) or EAX and EDX (64-bit results). Strings (and the like) are returned by reference in EAX
- Parameters passed to procedures are pushed onto the stack in *reverse* order. Example: For function `MyFunc(foo, bar, bas)`, `bas` is pushed onto the stack first, then `bar` and `foo` last
- Procedures will not remove parameters from the stack, the caller must do that after the procedure returns

GNU/Linux x86-64 Calling Conventions (Simplified)

In contrast to 32-bit mode, arguments are often passed in registers (not on the stack)

- Integers and addresses (e.g. references to strings)
- Order of registers: RDI, RSI, RDX, RCX, R8 and R9
- Remaining arguments are passed on the stack

More complex types are passed on the stack

- Stack must be aligned on a 16-byte boundary

Return values are in RAX and eventually RDX

Refer to *System V AMD64 ABI*[5] for details

[5] https://raw.githubusercontent.com/wiki/hjl-tools/x86-psABI/x86-64-psABI-1.0.pdf

Function `puts()`

`puts()` writes a string to `stdout`

- Similar to the sys_write `syscall`
- Requires a *single* argument (RDI): the address of the string to write
- The string must be terminated with a 0-byte (a C string!)

See `examples/` directory

Function `printf()`

The C function `printf()` - print and format

- Also writes a string to `stdout`
- Replaces place holders with arguments (e.g. variables)
- Provides a wide range of formatting options

Example

- The following writes “`2 + 3 = 5`”:

```
printf("%d + %d = %d", 2, 3, 5);
```

Format specifiers (see “`man 3 printf`” for more)

- `%c` character
- `%d` integer in decimal
- `%s` string
- `%x` integer in hexadecimal
- `%%` a percent symbol

See `examples/` directory

Function `scanf()`

The C function `scanf()`

- Reads-in a string corresponding to a given format from `stdin`
- First argument is the address of a string specifying the format
- Followed by the address(es) where the data which has been read should be stored

Example

- The following instructions reads-in an integer value (whole number):

```
int i;
scanf("%d", &i);
```

Format specifiers (see "`man 3 scanf`" for more)

- Similar to `printf`

See `examples/` directory

Conclusion

Conclusion

Procedures

- Useful for structuring a program
- Beware of caller-saved and callee-saved registers

Example: `hexdump2`

- Provides a hexdump like `hexdump1`
- Additional ASCII output
- Structured using procedures

Calls to the C standard library

- Requires linking with `gcc`
- Different entry point (`main`)
- Parameters are written in specific registers or on the stack
- Terminate the program with `ret` instead of `syscall`

Libraries

- For procedures to be reused
- Test and debug once, reuse many times