B35APO: Computer Architectures

Lecture 10. Function Calls and C Language

Pavel Píša
pisa@fel.cvut.cz

Petr Štěpán
stepan@fel.cvut.cz

CZECH TECHNICAL UNIVERSITY IN PRAGUE

27. června, 2025

Outline

1 C Language to Machine Code Translation

2 RISC-V Calling Convention – Defined by ABI

3 Operating System Services – System Calls

Today's Lecture Objective

- Find out how a C program is translated into machine instructions (RISC-V ISA as an example)
- Main focus on how a function calls are translated
- Where local and global variables are stored
- Calling operating system functions differs from calling functions

The Steps to Translate C Code to Machine One

A simple example – assignment translation `a = b + c;`

1. Assign registers to variables, e.g. b - t0, c - t1, a - t0
2. Load variable values into registers:

   ```
   lw t0, &b(gp);
   lw t1, &c(gp)
   ```
3. Perform a calculation:

   ```
   add t0, t0, t1
   ```
4. Store a value in a variable `a`:

   ```
   sw t0, &a(gp)
   ```

- Expressions are analyzed using context-free grammar - it covers all expressions possible in the C language.
- The example is without optimization, the value i simediatelly stored to memory even that it can be required and loaded again.
- If the addresses of variables were not reachable by a register gp relative addressing, then it would be necessary to load variable address into register before `lw` instruction.

While Loop Translation

More complex example – translation of `while (cond) body;`

- analysis of the context-free grammar of the language detects the while construct
- the condition expression `cond` is translated into instruction sequence `COND` recursively and then cycle `body` is translated to the `BODY` sequence.
- the instruction `j cond_1` is generated to jump to `cond_1` label
- `body_1` label is inserted
- `BODY` instructions sequence is inserted
- `cond_1` label is inserted
- `COND` instruction sequence is inserted, it stores result to the register, i.e. `t0`
- then the conditional branch is generated `bne t0, zero, body_1`

```
  j  cond_1
body_1:
  BODY
cond_1:
  COND
  bne t0, zero, body_1
```

Why Bother with Translation/Compilation?

- Either interpreted languages are used, but they are inherently slower
  - They can be faster only if they use libraries translated into machine instructions and are usually highly optimized, e.g. OpenCV, NumPy, TensorFlow, PyTorch, etc.
- or compiled languages are used, examples: C/C++, Rust, Fortran, Pascal and program is not directly executed in C, it is translated into machine instructions.
- If programmers have no idea what is result of translation:
  - they rely fully on compiler optimizations and can be surprised
  - defining a local variable/array of 100MiB size in function can be a problem and is no go for multithreaded programs
  - can overflow stack when recursion is used but depth of recursion is not known
- In homework 4, you will practice machine code analysis and try to write a C program that will behave similarly (the ideal goal would be to compile it into the specified or more optimized machine code).

How Is the Function Call Translated?

To compile a function, e.g. `int addtwo(int a, int b);`, next questions has to be resolved:

- How will arguments `a`, `b` be passed?
- How will be returned result of `addtwo`?
- How to finalize the function, which next instruction should be executed?

The application binary interface (ABI) of caller and callee must match to allow the correct behavior.

- The compilation of the callee can be on a different computer, by a different compiler (typically a library) than the compilation of the caller - your compiler on your computer.
- It is necessary to define a convention, mentioned ABI and all object code units and libraries have to match. The ABI is stored in object files and even final executables to allow check that they match.

Outline

1 C Language to Machine Code Translation

2 RISC-V Calling Convention – Defined by ABI

3 Operating System Services – System Calls

RISC-V Calling Convention – Defined by ABI

- Parameter values are stored to registers a0, ... , a7
  - If more than eight words in arguments are required then memory is used, see latter.
- The result of the function is stored to a0 and a1 registers.
  - If the result exceeds two words/registers then memory is used.
  - Caller has to reserve memory to store the result.
  - The pointer to that memory location is passed as the hidden first argument to the function.
  - Callee stores result directly to the location pointed by the register.

```c
struct a {
  int a, b, c, d;
};

struct a permut(int x, int y);

struct a t;

t = permut(2, 3);
```

```c
struct a {
  int a, b, c, d;
};

void permut(struct a *r, int x, int y);

struct a t;

permut(&t, 2, 3);
```

How to Return from a Function to the Right Caller?

```
[0x100]  j  addtwo

          ...

[0x254]  j  addtwo


addtwo:

     ...

     j ?
0x104 or 0x258?
or even somewhere else
```

- The `addtwo` function is caller from many different locations in the program
- it is not possible to fill address of the final/return jump at compile time
- the restur address has to be set by caller
- function call convention (ABI) – the return address is stored in the `ra` (return address) registr (`x1`)
- including of an instruction to jump to the address stored in the register is necessary
- an instruction which stores next instruction address into `ra` before branch would help a lot as well

JAL, JALR (RET, JR) Instructions

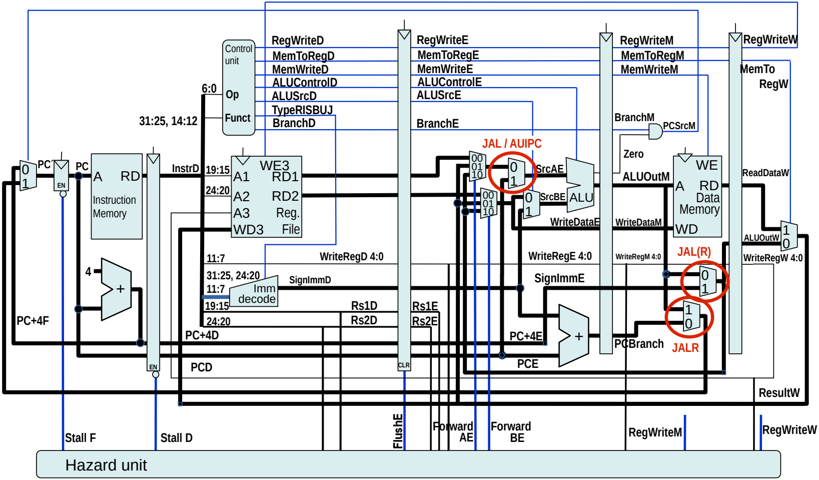
`jal` *`rd`*`, address`

- jump to address and store PC+4 into register *rd*
- if *rd* is not specified, then `x1` is substituted
- if only one directional jump is requested (instruction `j`), then PC+4 is discarded, `x0` is encoded as *rd* with `jal` opcode
- the target address is encoded into instruction as 21-bit signed extended offset to PC/the `jal` instruction location, LSB is 0, the limit of the target range is ±1 MB

`jalr` *`rd`*`,` *`rs1`*`, imm12`

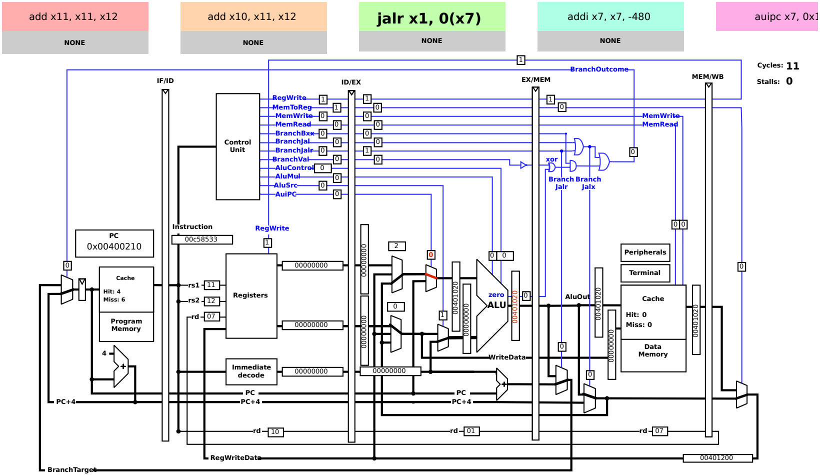
- jump to address *rs1* `+ imm12`×2 and store PC+4 into *rd* register
- if *rd* is not specified, then default `x1` is used for `jalr`
- if only return from the function is requested then `x0` register is used as *rd* which result in the same code as instruction alias `jr`

JAL and JALR Implementation

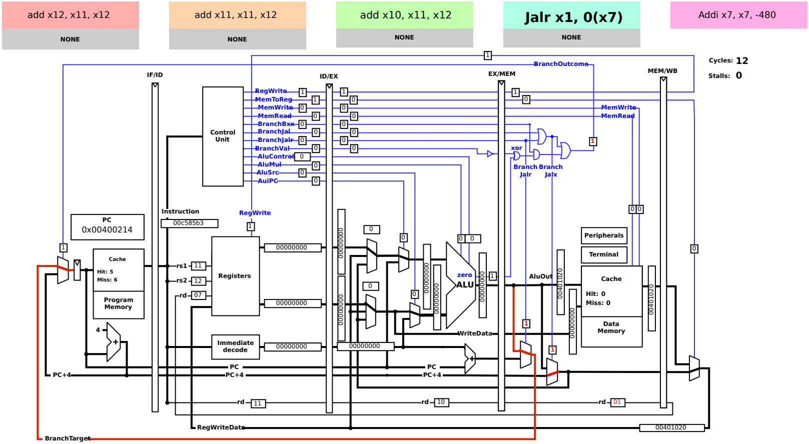


Quiz: How many of the connections are you able to describe their use
A – none B – about one third C – about half D – almost all.

JALR in QtRvSim Simulator



JALR – Control Hazard Caused by Jump



What to Do If the Function Calls Another One in Its Body?

The caller prepares the arguments in registers `a0` to `a7` and the return address in register `ra`.

But how to solve, keep argument values, when a function calls another function in its body?

It is necessary to store the register `ra` and registers `a0`–`a7` somewhere, but where?

- Activation record or activation frame is used to store function temporary variables over another function call.
- This record, or frame, is stored on the stack (call stack or stack frame).

Stack

Stack is LIFO (Last In First Out) data stucture – the last stored value is retrieved as the fisrt

- push – store data onto stack top, stack is one element deeper
- pop – take value from the top, previous one is on top now

Stack implementation where the top is defined by `sp` (`x2`) register:

- push

```
addi sp, sp, -4     – space allocation
sw   x10, 0(sp)     – value store
```

- pop

```
lw   x10, 0(sp)     – value restore
addi sp, sp, 4      – space release
```

push(1)

push(2)

push(3)

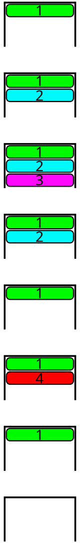
pop() == 3

pop() == 2

push(4)

pop() == 4

pop() == 1



What Is Stored onto Stack in Function?

Activation or stack frame stores:

- Return address and arguments which are used latter when another function is called.
- Function local variables, which lifetime is limited by the function body execution.
- The ABI specification forbids to return from function with any of `s0`-`s11` modified from state at the call
    - `s0` register is sometimes used as the pointer to start of an activation frame – frame `fp` pointer
    - `fp` register points to the fixed stack position during function body and this allows use it in relative addressing of local variables, arguments stored on stack and for final stack restoring if `alloca` and or dynamic length local arrays are used and allocated by `sp` register advances.

RISC-V Calling Convention – Part of ABI

Which registers can be freely modified (clobbered) by callee and which registers values have to be preserved to be returned the same to caller.

Mnemonic name	Register	Role in function	Calee can clobber
zero	x0	fixed zero/discard	– –
a0 - a7	x10 - x17	function input arguments	yes
a0, a1	x10, x11	return value/result	yes
ra	x1	return address	no
t0 - t6	x5-7, x28-x31	temporary/automatic variables	yes
s0 - s11	x8-9, x18-x27	saved registers	no
sp	x2	stack pointer	no
gp	x3	global (variables) pointer	– –
tp	x4	thread pointer/thread local store	– –

32-bit RISC-V Calling Convention – Argument Type Rules

- char, short, int, long, float, pointer – each argument in single register
- long long int, double – each argument in two registers (the first one aligned to even number)
- the content of structure passed by value is copied into as many registers as necessary
- if code is compiled for the RISC-V with floting point extension, then fload and double arguments are passed in FPU registers `f10` – `f17`, menmonic designation `fa0` – `fa7`
- if the arguments do not fit into appropriate argument registers then the rest is passed on the stack in the new function stack frame prepared for callee
- when the function is called (`jal`/`jalr` executed), the stack has to be 16 bytes aligned

Quiz

When are some of the registers `s0`-`s11` stored onto stack?

A never.

B when they will be clobbered by calling of another function in the function body.

C if they are used for function local variables or `s0` as `fp`.

D unconditionally each time when function is called.

Quiz

When are some of the registers a0-a7 stored onto stack?

A never.

B if there are so many local variables in the function, that they do not fit into registers.

C if the another function is called in the function body and given input argument are used in computation after that call .

D only if the function is recurrent (calls itself recursively).

Function Calling

The function calling with max 8 parameters:

```
t = addfour(1, 2, 3, 4);
```

Compiled for RISC-V

```
li  a3,4
li  a2,3
li  a1,2
li  a0,1
jal ra,10054 <addfour>
```

Calling of the function with 10 arguments:

```
t = addten(1, 2, 3, 4, 5,
           6, 7, 8, 9, 10);
```

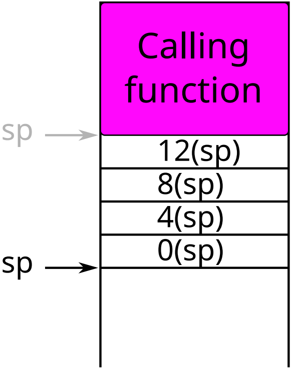
Compiled for RISC-V

```
li  a5,10
sw  a5,4(sp)
li  a5,9
sw  a5,0(sp)
li  a7,8
li  a6,7
li  a5,6
li  a4,5
li  a3,4
li  a2,3
li  a1,2
li  a0,1
jal ra,10054 <addten>
```

Function Translation – Simple Function

Example with few parameters without internal function call with four local variables

- Space for local/automatic variables is allocated on the stack
  `addi sp,sp,-16`
- Local variables are accessible on the stack: 0(sp), 4(sp), 8(sp), 12(sp)
  - If there is enough available registers then stack is not for leaf-node function used at all



Function finalization:

- Release/free space for local variables
  `addi sp,sp,16`
- Return to the caller after `jal/jalr`
  `jalr 0(x1)` – `ret`

The Function with 10 Arguments and Call in the Body

The function allocates space for two local variables in addition.

Function prologue/start:

```
addi  sp,sp,-48
sw  ra,44(sp)
sw  s0,40(sp)
sw  s1,36(sp)
sw  s2,32(sp)
sw  s3,28(sp)
sw  s4,24(sp)
```

Access to the local variables:

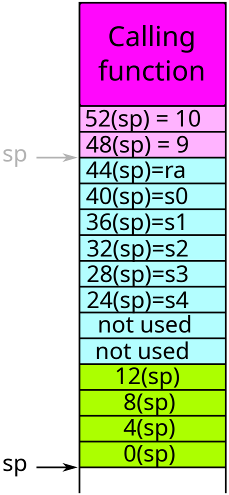
```
lw  t0,8(sp)
lbu t1,15(sp)
```

int i; location 8(sp)
char c; location 15(sp)
0(sp) – 7(sp) unused or reserved for inner call argumens

Function finalization/epilogue:

```
lw   ra,44(sp)
lw   s0,40(sp)
lw   s1,36(sp)
lw   s2,32(sp)
lw   s3,28(sp)
lw   s4,24(sp)
addi sp,sp,48
ret
```

Remark: the stack allocations are aligned to 16 bytes.



Calling function
52(sp) = 10
48(sp) = 9
sp →
44(sp)=ra
40(sp)=s0
36(sp)=s1
32(sp)=s2
28(sp)=s3
24(sp)=s4
not used
not used
12(sp)
8(sp)
4(sp)
0(sp)
sp →

Function Frame Pointer

- The function frame pointer contains the value of the `sp` at function entry
- The `s0` register is used as `fp` (frame pointer), it is callee save (non-clobberable) register, which has to be saved at function entry and restored at exit.
- Advantages of `fp` use:
  - Arguments and local variables of the function addressable by fixed offsets to `fp` even if `sp` is changed inside function body (e.g. arguments for function calls) which requires offsets to `sp` recalculations or make it impossible in `alloca` case
  - The stack frames unwinding is much easier, i.e. for debugging
    - Stack unwinding is required even in case of exception processing in C++ when the exception is catch in same function in the caller chain to the throwing function
    - It is necessary to restore stack into state which corresponds its state inside catching caller function at time of the call of the function leading to the one causing the exception.
- Disadvantages of `fp` use:
  - Slows down the program, although it is only a few instructions when entering and exiting the function, but if the function is called often and its body is short, it can be a significant overhead.
  - `fp` occupies the register `s0`, which could be used to store something else. If there are no free registers, then the value must be stored on the stack in RAM (via cache), which is slower than using the register.

Compile Function with Enforced Frame Pointer

The GCC compiler switch `-fno-omit-frame-pointer` is used to enforce frame pointer

Prologue of the function in RISC-V case

```
addi  sp,sp,-48
sw    ra,44(sp)
sw    s0,40(sp)
sw    s1,36(sp)
sw    s2,32(sp)
sw    s3,28(sp)
sw    s4,24(sp)
sw    s5,20(sp)
addi  s0,sp,48
```

Access to the local variable:

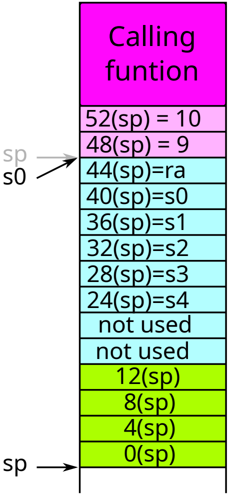
```
sw a0,-36(s0)
```

in the case without `fp`:

```
sw a0, 12(sp)
```

Function finalization:

```
lw   ra,44(sp)
lw   s0,40(sp)
lw   s1,36(sp)
lw   s2,32(sp)
lw   s3,28(sp)
lw   s4,24(sp)
lw   s5,20(sp)
addi sp,sp,48
ret
```



Calling funtion

52(sp) = 10

48(sp) = 9

44(sp)=ra

40(sp)=s0

36(sp)=s1

32(sp)=s2

28(sp)=s3

24(sp)=s4

not used

not used

12(sp)

8(sp)

4(sp)

0(sp)

The Homework No. 4 – Subroutine

Analyze (reverse engineering source code) of function `subroutine_fnc`:

- find which of a0 – a7 are used in function and are filled by arguments before its call
    - WARNING: compiler uses aX registers as temporal variables for computation same as t0 – t6 registers, their values can be changes without care about caller
- find meaning of return value in a0 when `ret` (`jalr 0(x1)`) instruction is reached
- analyze prologue of the function:
    - if it starts by instructions sequence similar to

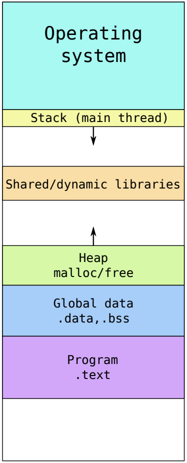
```
addi  sp,sp,-48
sw  ra,44(sp)
sw  s0,40(sp)
sw  s1,36(sp)
sw  s2,32(sp)
sw  s3,28(sp)
sw  s4,24(sp)
```

        - then function uses stack to store return address (`ra`) and local variables or makes sX available for local variables
        - locations `0(sp)` to `23(sp)` can be used for local/automatic variables
        - s0 – s4 are used for local variables in this case, usually to keep arguments over other function or system calls
    - if the function does not contain `addi sp,sp,-X` then it is a "simple" function which does not use stack
        - all local/automatic variables and intermediate calculations are stored in unused a0 – a7 and t0 – t6 registers

Process Address Space Organization – 32-bit Example

Each process has it own memory context with 4GiB of virtual address space:

- The operating system reserves the top 1GiB for itself (3GiB limit)
- Stack is below the limit (if the process has multiple threads, then the thread stacks are allocated from heap)
- Below is space used to map dynamic libraries
- Dynamically allocated memory grows from global `.data`+`.bss` end – heap (malloc,new/free,delete)
- Global data (initialized `.data` and uninitialized `.bss`)
- Program (.text)
- Some range above 0 is left unmapped to catch NULL pointer dereference errors.



The Homework No. 4 – Global Variables

Global variables presence indication:

- A global variable is a variable defined outside a function, or inside a function using the `static` keyword
- Global initialized variables are placed in the `.data` section, uninitialized (but zeroed in the C case) variables in the `.bss` section
- How do I find out if my program has global variables?
  - look at the end of the RISC-V assembler listing and find the `my_data` section.

    This section might look like this:

    ```
    Contents of section my_data:
     11008 00000000                    ....
    ```

    - 11008 is the address in the process virtual address space
    - 00000000 is hexadecimal listing/representation of the stored data (its initial value)
    - .... are values displayed as ASCII representation, if the character is unprintable, the dot is on given position
    - if the global variable/data is used inside code then the access can look as:

      ```
      lui  a5, 0x11
      addi a5, a5, 8 # 11008
      lw   a4, 0(a5)
      ```

Quiz

Consider the function below

```
int fce (int a) {
  int i;

  // function body

  return i+a;
}
```

Where are argument a and variable i stored in the RISC-V case:

- A both on the stack or in registers
- B both in the data section
- C a on the stack or in register, i in the data section
- D a in the data section, i on stack or in register

Security Vulnerability – Attacking a Program via the Stack

Consider following program:

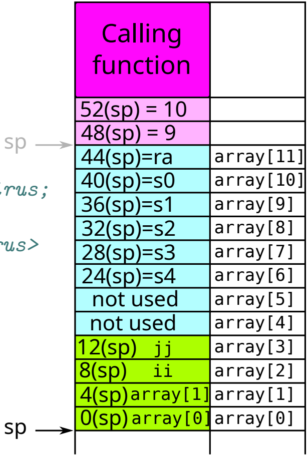
```
int virus() {
  // attaceker code
  return 0;
}

int addnum(int a, int b, int c,
  int d, int e, int f, int g,
  int h, int i, int j) {

  volatile int ii,jj=i+j;
  volatile int array[2];

  // some computations
  // function calling

  array[11] = (int)&virus;

  return array[0]+array[1];
}
```

The function prologue:

```
addi sp,sp,-48
sw   ra,44(sp)
sw   s0,40(sp)
sw   s1,36(sp)
sw   s2,32(sp)
sw   s3,28(sp)
# array[11]=(int)&virus;
lui  a5,0x10
addi a5,a5,84 # <virus>
sw   a5,44(sp)
```

Function epilogue:

```
lw   ra,44(sp)
lw   s0,40(sp)
lw   s1,36(sp)
lw   s2,32(sp)
lw   s3,28(sp)
addi sp,sp,48
ret
```



Calling function	
52(sp) = 10	
48(sp) = 9	
44(sp)=ra	array[11]
40(sp)=s0	array[10]
36(sp)=s1	array[9]
32(sp)=s2	array[8]
28(sp)=s3	array[7]
24(sp)=s4	array[6]
not used	array[5]
not used	array[4]
12(sp) jj	array[3]
8(sp) ii	array[2]
4(sp)array[1]	array[1]
0(sp)array[0]	array[0]

Variadic Functions in C – Arguments Access with va_list

Definition of the function with variable arguments count:

```c
#include <stdarg.h>

int sum_n_args(int n, ...) {
  int sum = 0;
  int i;
  va_list ap;

  va_start(ap, n);
  for (i=0; i<n; i++) {
    sum+=va_arg(ap,int);
  }
  va_end(ap);
  return sum;
}
```

Function calling:

```c
int  main() {
  printf("Sum %d\n", sum_n_args(10, 1,2,3,
                        4,5,6,7,8,9,10);
  printf("Sum %d\n", sum_n_args(2, 1,2);
  printf("Sum %d\n", sum_n_args(8, 1,2,3,
                              4,5,6,7,8);
}
```

Translation of the Var. Arg. Function sum_n_args

Preprocessor macros va_start and va_arg requires that all arguments are stored in the memory:

Function prologue:

```
addi sp,sp,-64
sw   ra,28(sp)
sw   s0,24(sp)
sw   s1,20(sp)
sw   a1,36(sp)
sw   a2,40(sp)
sw   a3,44(sp)
sw   a4,48(sp)
sw   a5,52(sp)
sw   a6,56(sp)
sw   a7,60(sp)
```

va_start(ap, n):

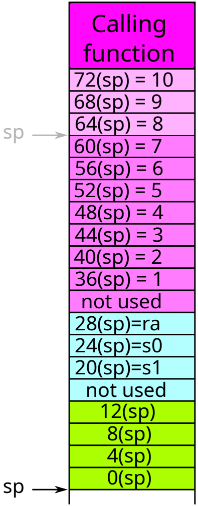
```
addi a5,sp,36
sw   a5,8(sp)
```

va_arg(ap, int):

```
lw   a4,8(sp)
addi a3,a4,4
sw   a3,8(sp)
lw   s0,0(a4)
```

Function epilogue:

```
lw   ra,28(sp)
lw   s0,24(sp)
lw   s1,20(sp)
addi sp,sp,64
ret
```



Calling function
72(sp) = 10
68(sp) = 9
sp → 64(sp) = 8
60(sp) = 7
56(sp) = 6
52(sp) = 5
48(sp) = 4
44(sp) = 3
40(sp) = 2
36(sp) = 1
not used
28(sp)=ra
24(sp)=s0
20(sp)=s1
not used
12(sp)
8(sp)
4(sp)
0(sp)
sp →

Outline

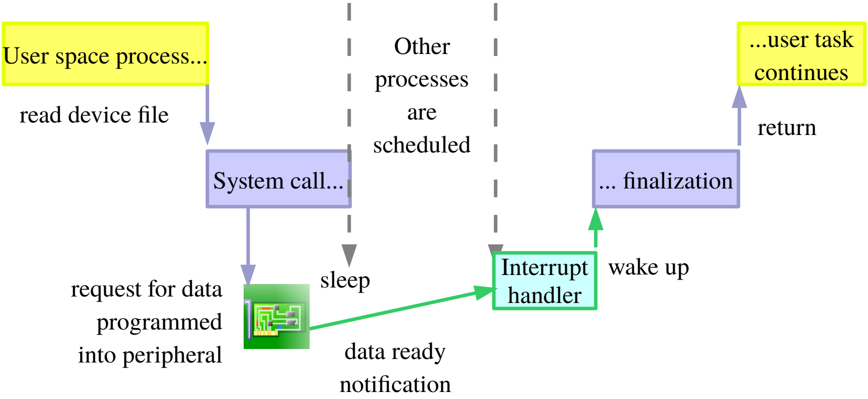
1. C Language to Machine Code Translation
2. RISC-V Calling Convention – Defined by ABI
3. Operating System Services – System Calls

Basic of Operating System Protection Mechanisms

- A user process cannot directly access the computer's HW.
- A user process must ask the OS to make the HW available or to handle a request.
- OS offers services with HW privileged access to processes in form of system calls.
- Unlike a function call, we do not know the addresses of functions in the kernel.
- A system call function is selected by the system call number.
- The actual function call is realized through an interrupt or exception with a specialized instruction:
  - x86 old – uses int 0x80 directly – invoke interrupt number 0x80.
  - x86 newer – specialized instructions sysenter/syscall – direct, no interrupt gates and accesses less memory → is faster.
  - RISC-V – specialized instruction ecall – invokes exception.
- Interrupt/exception is handled by the service routine starting at preconfigured address (i.e. `mtvec`/`stvec`) in priviledged mode (system or machine) and the user cannot change it.

Device Input Realized by System Calls

The user program prepares the call parameters and system call number in the registers and triggers a special interrupt/exception (`ecall` on RISC-V). The OS calls the corresponding function based on the system call number.



Return from system call to user program uses same or similar mechanism as return from interrupt/exception (usually `sret` on RISC-V).

Application Binary Interface (ABI) – System Calls

- API – application programming interface – standard, which function can program call program from libraries.
    - API is defined for the C language by header files.
    - API also defines what the given functions do, what are their return values, how they behave in error case
    - try e.g. `man 2 read`
- ABI – application binary interface – description of which registers and which instructions to use
- ABI for system calls on the RISC-V architecture
    - a7 – contains the system call number (for an overview, e.g. `https://jborza.com/post/2021-05-11-riscv-linux-syscalls/` or `https://marcin.juszkiewicz.com.pl/download/tables/syscalls.html`)
    - a0 to a5 – system call parameters (Linux system calls have a maximum of 6 parameters)
    - a system call is made with the `ecall` instruction
    - a0 contains the return value of the system call
        - if the call should return more data (e.g. reading from a file), the user must specify a pointer to the buffer and the size of the buffer where the OS will write the data.

Hello World! Example for RISC-V and Linux kernel

```
.global _start
.text
_start:
#   write(1, "Hello world!\n", 13);
   addi  a7, zero, 64
   addi  a0, zero, 1
   la    a1, zero, text_1
   addi  a2, zero, 13
   ecall
final:
#   exit(0);
   addi  a7, zero, 93
   addi  a0, zero, 0
   ecall
   ebreak
   j final
.data
# store ASCII text, no termination
text_1: .ascii "Hello world!\n"
```

The Homework No. 4 – System Calls

How to identify and analyze system calls?

- locate `ecall` instructions.
- determine value set in `a7` before `ecall`, it specifies which system service is requested.
- find out the values of registers `a0`, `a1`, `a2` (or `a3` if used).
- find out corresponding function prototype in the C runtime library and combine it with found arguments.
- Example: you find out that register `a7` has the value 63 – `read`, re-implement code as the system library function call
  `read(a0, a1, a2);`
  - the only problem is with the function open/openat, whose `O_`*XXXX* flags have different values for the x86 system and for RISC-V
  - because programs are checked by Brute on the x86 system, it is better to verify the values of the parameters in the dump program-x86.list

x86 Linux Kernel System Calls

- the system call is realized by the instruction `int 0x80`.
- the system call number is passed in the `eax` register.
  - ATTENTION the x86 and RISC-V system call numbers are different.
- system call parameters are stored sequentially in registers:
  - ebx
  - ecx
  - edx
  - esi
  - edi
  - ebp
- In the next lecture, x86 assembler will be described which allows to use x86 code variant listing to be used to solve homework 4.

Quiz

Consider the function bellow

```
int fce (int a) {
  static int s;
  int i;

  // funtion body

  return i+s;
}
```

Where are variables `s` and `i` located?

A both allocated on the stack

B both in the `.data` section

C `s` on the stack, `i` in the `.data` section

D `s` in the `.data` section, `i` on the stack