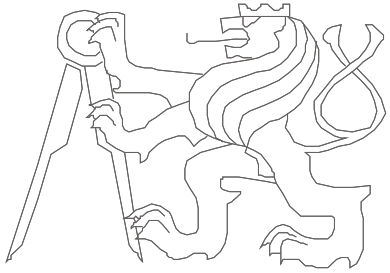
Architektura počítačů

Předávání parametrů funkcím a virtuálním instrukcím operačního systému



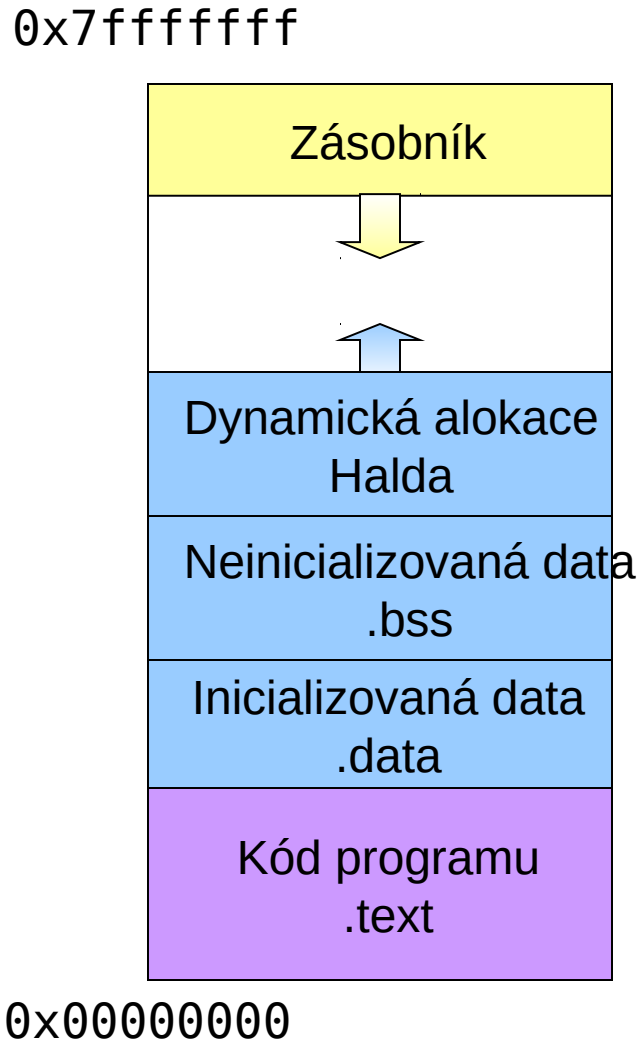
České vysoké učení technické, Fakulta elektrotechnická

Různé druhy volání funkcí a systému

- Volání běžných funkcí (podprogramů)
  - Možnosti předávání parametrů – v registrech, přes zásobník, s využitím registrových oken
  - Způsob definuje volací konvence – musí odpovídat možnostem daného CPU a je potřeba, aby se na ní shodli tvůrci kompilátorů, uživatelských a systémových knihoven

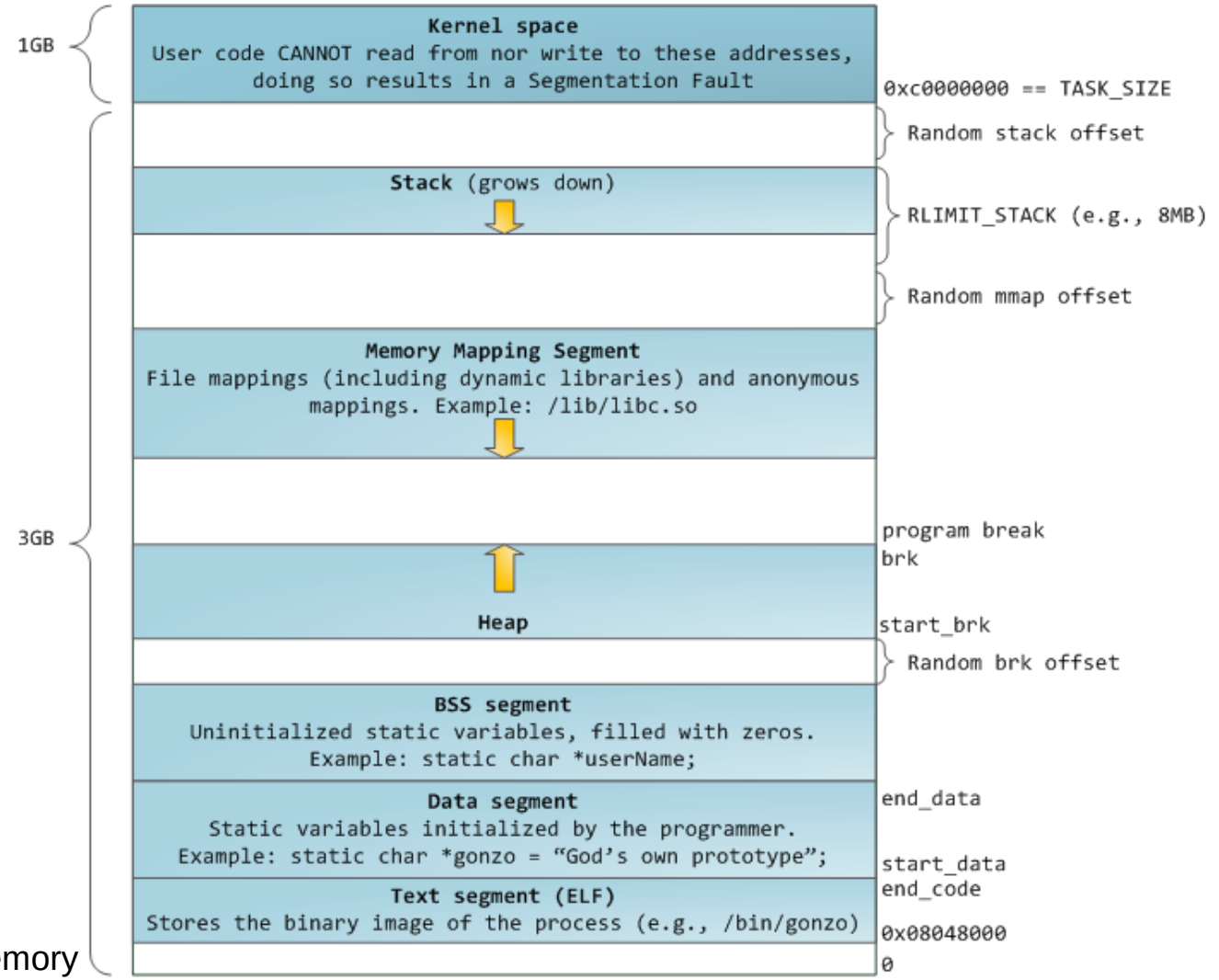
    (x86 Babylón - cdecl, syscall, optlink, pascal, register, stdcall, fastcall, safecall, thiscall MS/others)
  - Zásobníkové rámce pro uložení lokálních proměnných a alloca()
- Volání systémových služeb
  - Přechod mezi uživatelským a systémovým režimem
- Vzdálená volání funkcí a metod (volaný nemůže číst paměť)
  - Přes síť (RPC, CORBA, SOAP, XML-RPC)
  - Lokální jako síť + další metody: OLE, UNO, D-bus, atd.

Rozložení programu ve virtuálním adresním prostoru



- Do adresního prostoru procesu je nahraný – mapovaný soubor obsahující kód a inicializovaná data programu – sekce **.data** a **.text** (možnost rozdílné LMA a VMA)
- Oblast pro neinicializovaná data (**.bss**) je pro C programy nulovaná
- Nastaví se ukazatel zásobníku a předá řízení na startovací adresu programu (**_start**)
- Dynamická paměť je alokovaná od symbolu **_end** nastaveného na konec **.bss**

Adresní prostor procesu (32-bit Linux)



Převzaté z Gustavo Duarte:
Anatomy of a Program in Memory
http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory

Postup volání podprogramu

- Volající program vypočítá hodnoty parametrů
- Uloží proměnné alokované do registrů, které mohou být volaným podprogramem změněné (clobberable register)
- Parametry jsou uloženy do registrů a nebo na zásobník tak jak definuje použitá volací konvence (ABI)
- Řízení se přesune na první instrukci podprogramu, přitom je zajištěné, že návratová adresa je uložena na zásobník nebo do registru
- Podprogram ukládá hodnoty registrů, které musí být zachovány a sám je chce využít (callee saved/non-clobberable)
- Alokuje oblast pro lokální proměnné
- Provede vlastní tělo podprogramu
- Uloží výsledek do konvencí daného registru(ů)
- Obnoví uložené registry a provede návrat na následující instrukci
- Instrukce pro návrat může v některých případech uvolnit parametry ze zásobníku, většinou je to ale starost volajícího

Příklad: registry a volací konvence MIPS

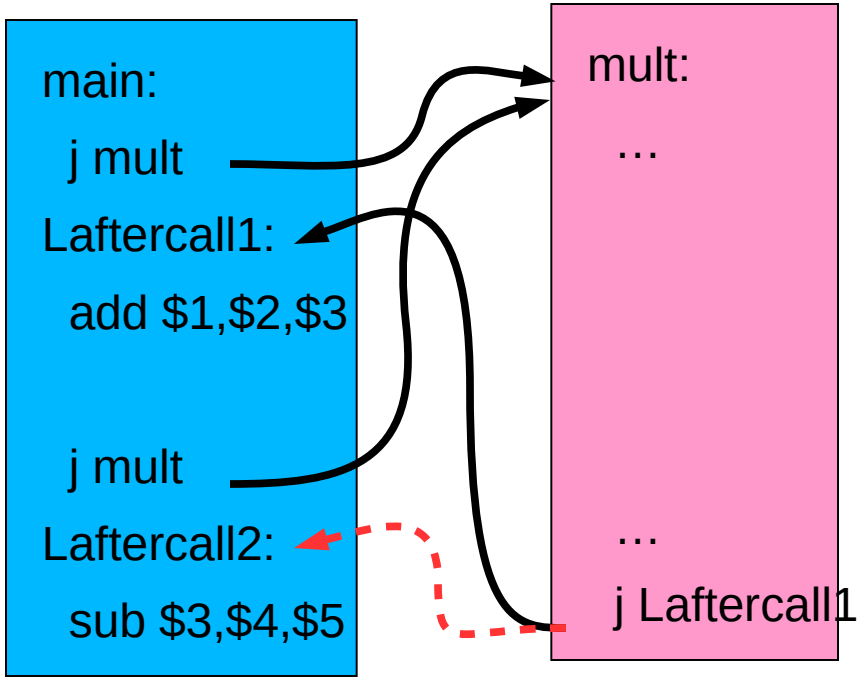
- a0 – a3: argumenty (registry $4 – $7)
- v0, v1: výsledná hodnota funkce ($2 a $3)
- t0 – t9: prostor pro dočasné hodnoty ($8-$15,$24,$25)
  - volaná funkce je může používat a přepsat
- at: pomocný registr pro assembler ($1)
- k0, k1: rezervované pro účely jádra OS ($26, $27)
- s0 – s7: volanou funkcí ukládané/zachované registry ($16-$23)
  - pokud jsou využity volaným, musí být nejdříve uloženy
- gp: ukazatel na globální statická data ($28)
- sp: ukazatel zásobníku ($29)
- fp: ukazatel na začátek rámce na zásobníku ($30)
- ra: registr návratové adresy ($31) – implicitně používaný instrukcí **jal** – jump and link

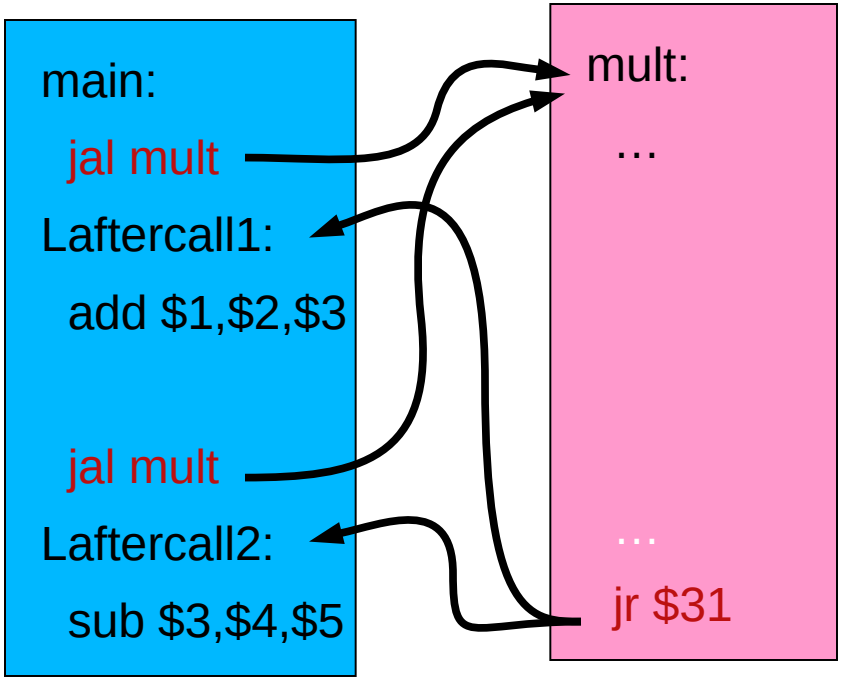
MIPS: instrukce pro volání a návrat

- Volání podprogramu: jump and link
  - **jal** ProcedureLabel
  - Adresa druhé (uvažujte delay slot) následující instrukce za instrukcí **jal** je uložena do registru **ra** ($31)
  - Cílová adresa je uložena do čítače instrukcí **pc**
- Návrat z podprogramu: jump register
  - **jr ra**
  - Obsah registru **ra** je přesunutý do **pc**
  - Instrukce je také použitelné pro skoky na vypočítanou adresu nebo adresu z tabulky – například konstrukce case/switch statements

Rozdíl mezi voláním a skokem

Skok neuloží návratovou hodnotu
kód tedy nejde využít z více míst





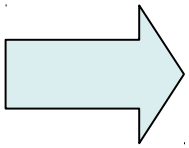
Autor příkladu Prof. Sirer
Cornell University

naopak volání s využitím registru **ra**
umožňuje volat podprogram tehdy,
kdy je potřeba

Instrukce Jump and Link podrobněji

C/C++

```
int main() {
  simple();
  …
}

void simple(){
  return;
}
```



MIPS

```
0x00400200 main: jal simple
0x00400204 …

0x00401020 simple: jr $ra
```

Popis:	For procedure call.
Operace:	$31 = PC + 8; PC = ((PC+4) & 0xf0000000) | (target << 2)
Syntax:	jal target
Encoding:	0000 11ii iiii iiii iiii iiii iiii iiii

Volající/**`jal`** uloží návratovou adresu do registru **`ra`** (reg $31) a běh programu se přesune na první instrukci volaného podprogramu. Podprogram je ukončený skoken na návratovou adresu (**`jr $ra`**).

Příklad kódu funkce

Zdrojový kód v jazyce C:

```
int leaf_example (int g, h, i, j)
  { int f;
    f = (g + h) - (i + j);
    return f;
  }
```

Parametry **g**, …, **j** jsou uložené v registrech **a0**, …, **a3**

Hodnota **f** je počítaná v registru **s0** (proto je potřeba **s0** uložit na zásobník)

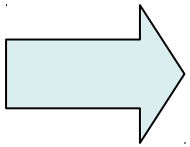
Návratová hodnota je uložená v registru **v0**

MIPS: Předání parametrů mezi volajícím a volaným

- $a0-$a3 hodnoty argumentů
- $v0-$v1 návratová hodnota(hodnoty)

C/C++

```
int main() {
  int y;
  y=fun(2,3,4,5)
  …
}

int leaf_fun(int a,
   int b, int c, int d)
{
  int res;
  res = a+b – (c+d);
  return res;
}
```



MIPS

```
main:
 addi $a0, $0, 2
 addi $a1, $0, 3
 addi $a2, $0, 4
 addi $a3, $0, 5
 jal fun
 add $s0, $v0, $0

fun:
 add $t0, $a0, $a1
 add $t1, $a2, $a3
 sub $s0, $t0, $t1
 add $v0, $s0, $0
 jr $ra
```

Volaný přepíše (Clobbers) registr garantovaný volanému Jako neměněný (saved) registr $s0

Příklad překladu pro architekturu MIPS

int leaf_example (int g, h, i, j)

g→$a0, h→$a1, i→$a2, j→$a3, $v0 – ret. val, $sX – save, $tX – temp, $ra – ret. addr

leaf_example:	
addi $sp, $sp, -4 sw $s0, 0($sp)	Save $s0 on stack

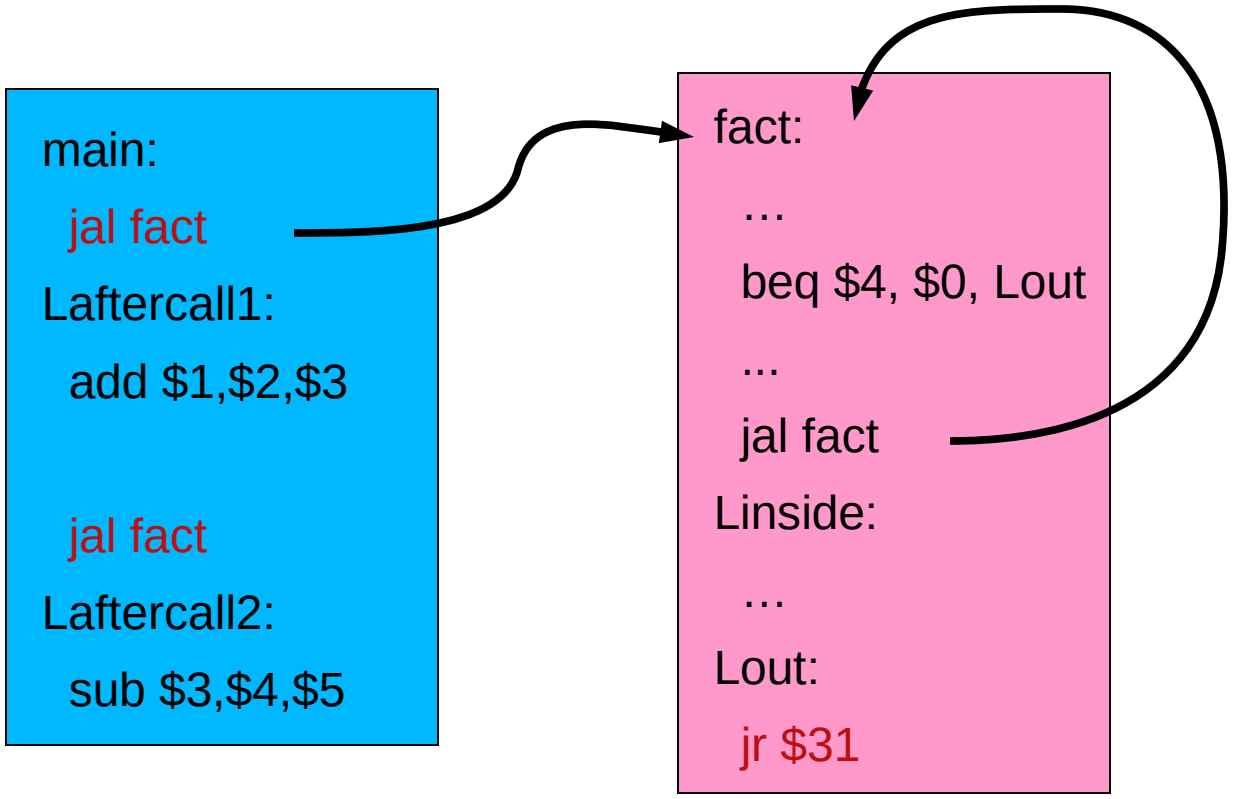
add $t0, $a0, $a1
add $t1, $a2, $a3
sub $s0, $t0, $t1
	Procedure body

add $v0, $s0, $zero	Result
lw $s0, 0($sp) addi $sp, $sp, 4	Restore $s0
jr $ra	Return

Delay-sloty nejsou uvažované, výpis před jejich plněním

Zdroj: Henesson: Computer Organization and Design

Problém vícenásobného volání s link-registrem



Registr **ra** je potřeba před voláním dalšího podprogramu (nebo rekurzivním voláním) někam uložit stejně jako jsou ukládané (saved/non-clobberable) registry **s***X*

Rekurzivní funkce nebo funkce volající další funkci

Zdrojový kód v jazyce C:

```
int fact (int n)
  {
    if (n < 1) return f;
    else return n * fact(n - 1);
  }
```

Parametr **n** je uložen v registru **a0**

Návratová hodnota v registru **v0**

MIPS - rekurzivní volání

```
fact:
    addi    $sp, $sp, -8        # adjust stack for 2 items
    sw      $ra, 4($sp)         # save return address
    sw      $a0, 0($sp)         # save argument
    slti    $t0, $a0, 1         # test for n < 1
    beq     $t0, $zero, L1
    addi    $v0, $zero, 1       # if so, result is 1
    addi    $sp, $sp, 8         #   pop 2 items from stack
    jr      $ra                 #   and return
L1: addi    $a0, $a0, -1        # else decrement n
    jal     fact                # recursive call
    lw      $a0, 0($sp)         # restore original n
    lw      $ra, 4($sp)         #   and return address
    addi    $sp, $sp, 8         # pop 2 items from stack
    mul     $v0, $a0, $v0       # multiply to get result
    jr      $ra                 # and return
```

Výpis před plněním/přesunem instrukcí do delay-slotů

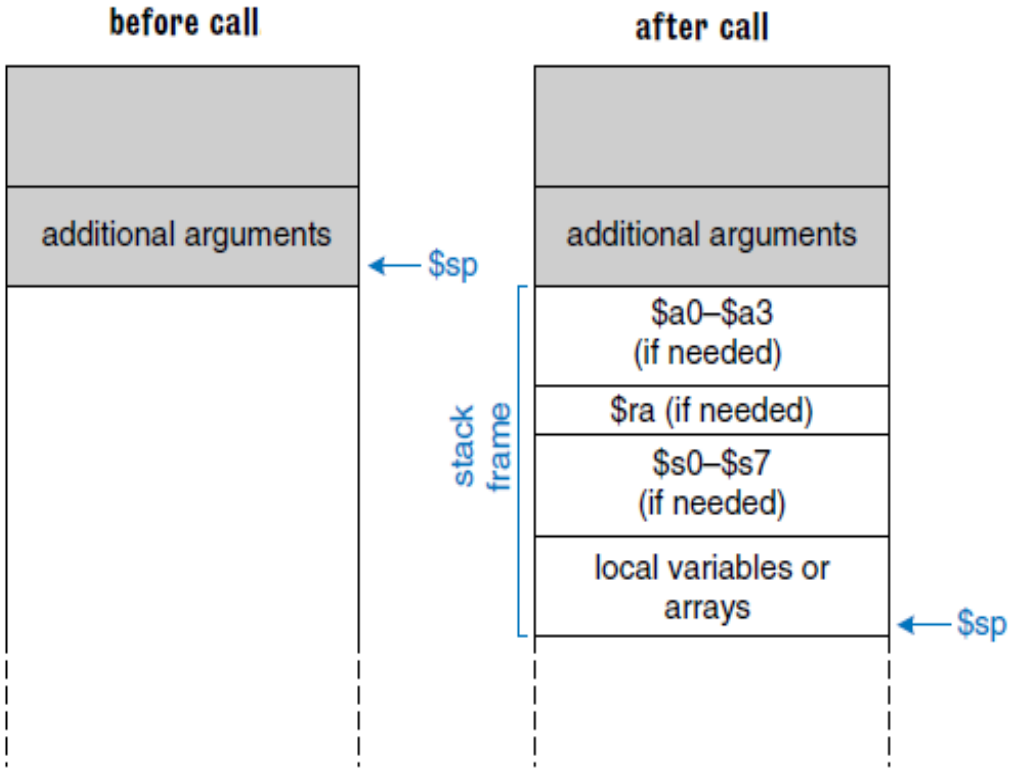
MIPS Calling Convention (ABI)

- Application binary interface (ABI) pro danou architekturu
- Definuje, které registry musí být uchovány volaným (callee)
- Specifikuje jak jsou předávané argumenty

Preserved by callee (Non-clobberable)	Nonpreserved (clobberable)
Callee-saved	Maintained/saved by caller
Saved registers: $s0 … $s7	Temporary registers: $t0 … $t9
Return address: $ra	Argument registers: $a0 … $a3
Stack pointer: $sp	Return value registers: $v0 … $v1
Stack above the stack pointer	Stack bellow the stack pointer

MIPS: podprogram s více než 4 argumenty

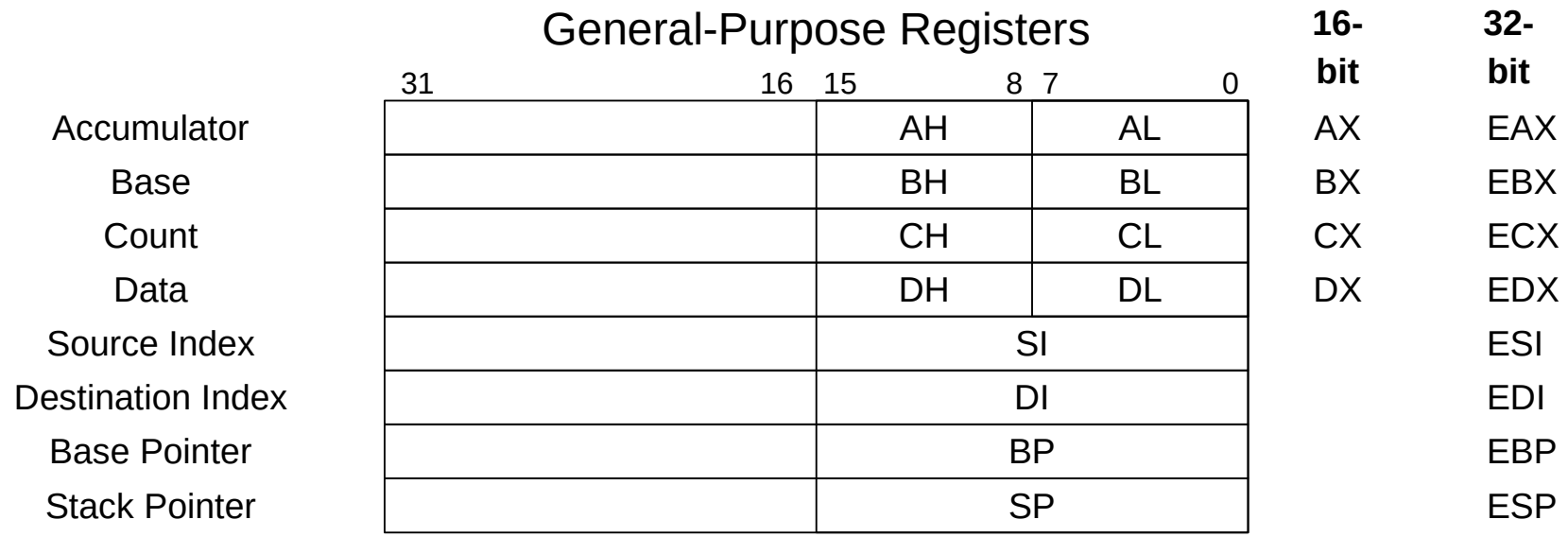
- MIPS calling convention: První čtyři $a0 … $a3, další uložené na zásobník tak, že pátý argument je uložený přímo na adrese $sp, další $sp+4. Prostor je alokovaný a uvolněný volajícím.

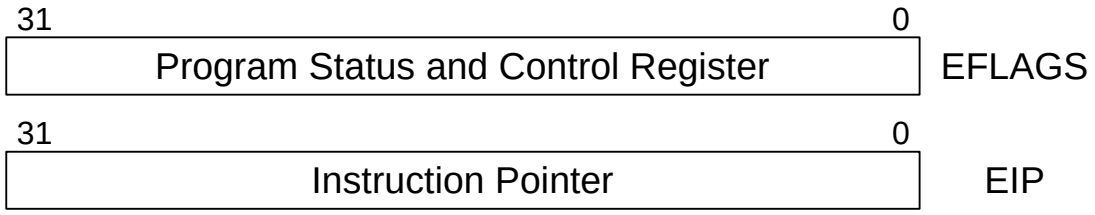


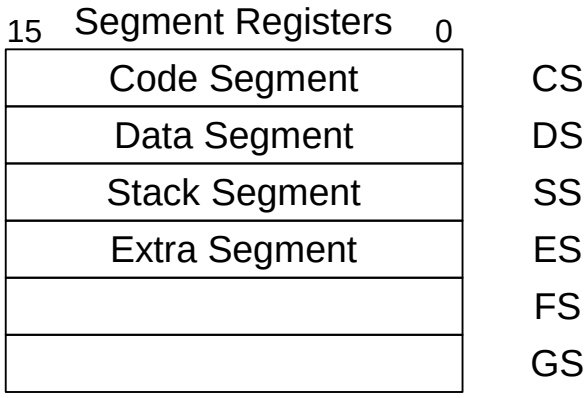

```
int main(){
  complex(1,2,3,4,5,6);
  return 0;
}
```

```
addiu  $sp,$sp,-8
addi   $2,$0, 5   # 0x5
sw  $2,4($sp)
addi   $2,$0,6    # 0x6
sw  $2,0($sp)
addi   $a0,$0,1   # 0x1
addi   $a1,$0,2   # 0x2
addi   $a2,$0,3   # 0x3
addi   $a3,$0,4   # 0x4
jal     complex
```

Volání C podprogramů v 32-bit módu x86 – registry







Volání C podprogramů v 32-bit módu x86 – assembler

GAS AT&T syntax	INTEL/MASM
movb $8, %al	mov al, 8
addw $0x1234, %bx	add bx, 1234h
subl (%ecx), %edx	sub edx, [ecx]
memory operands	memory operands
100	[100]
%es:100	[es:100]
(%eax)	[eax]
(%eax,%ebx)	[eax+ebx]
(%ecx,%ebx,2)	[ecx+ebx*2]
(,%ebx,2)	[ebx*2]
-10(%eax)	[eax-10]
%ds:-10(%ebp)	[ds:ebp-10]

Volání C podprogramů v 32-bit módu x86 – assembler

GAS AT&T

```
# Full example: load *(ebp + (edx * 4) - 8) into eax
movl    -8(%ebp, %edx, 4), %eax

 # Typical example: load a stack variable into eax
movl    -4(%ebp), %eax

# No index: copy the target of a pointer into a register
movl    (%ecx), %edx

# Arithmetic: multiply eax by 4 and add 8
leal    8(,%eax,4), %eax

# Arithmetic: multiply eax by 2 and add edx
leal    (%edx,%eax,2), %eax
```

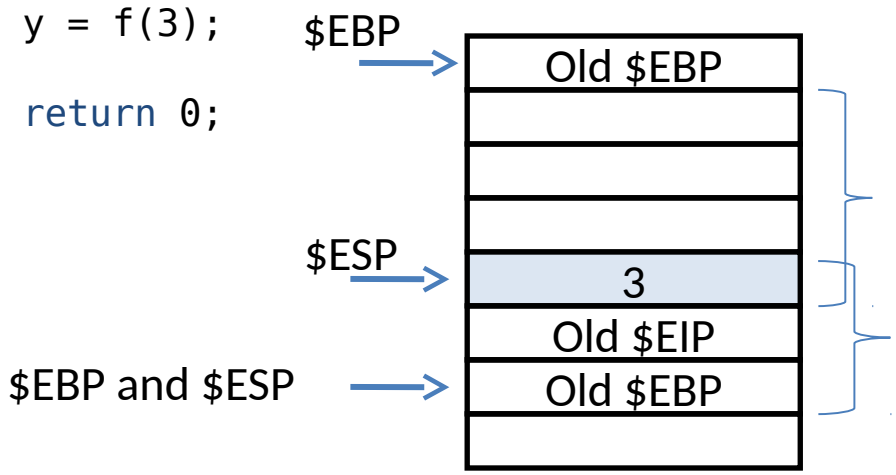
Běžné C volání x86 pro 32-bit režim

- Registry $EAX, $ECX, a $EDX mohou být podprogramem modifikované
- Registry $ESI, $EDI, $EBX jsou přes volání zachované
- Registry $EBP, $ESP mají speciální určení, někdy ani $EBX nelze použít obecně (vyhrazen pro GOT přístup)
- Tři registry jsou většinou nedostatečné i pro lokální proměnné koncových funkcí
- $EAX a pro 64-bit typy i $EDX je využitý pro předání návratové hodnoty
- Vše ostatní – parametry, lokální proměnné atd. je nutné ukládat na zásobník

SYSTEM V APPLICATION BINARY INTERFACE
Intel386™ ArchitectureProcessor Supplement

x86 – volání funkce s jedním parametrem

```c
#include <stdio.h>

int f(int x)
{
        return x;
}

int main()
{
        int y;

        y = f(3);

        return 0;
}
```




```
f:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %eax
        leave   ; same as
                ; mov %ebp,
   %esp
                ; pop %ebp
        ret

main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        subl    $16, %esp
        movl    $3, (%esp)
        call    f
        movl    %eax, -4(%ebp)
        movl    $0, %eax
        leave
        ret
```

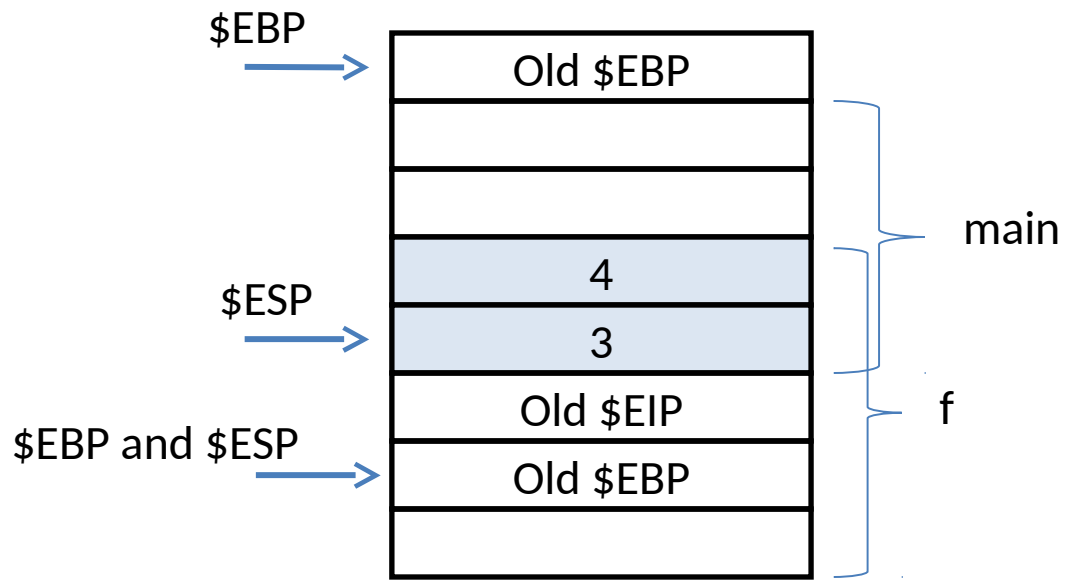
Příklady převzaté z prezentace od autorů David Kyle a Jonathan Misurda

x86 – volání funkce se dvěma parametry

```
#include <stdio.h>

int f(int x, int y)
{
  return x+y;
}
```

```
int main()
{
  int y;
  y = f(3, 4);
  return 0;
}
```

```
f:      pushl   %ebp
        movl    %esp, %ebp
        movl    12(%ebp), %eax
        addl    8(%ebp), %eax
        leave
        ret
main:   pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        subl    $16, %esp
        movl    $4, 4(%esp)
        movl    $3, (%esp)
        call    f
        movl    %eax, 4(%esp)
        movl    $0, %eax
        leave
        ret
```



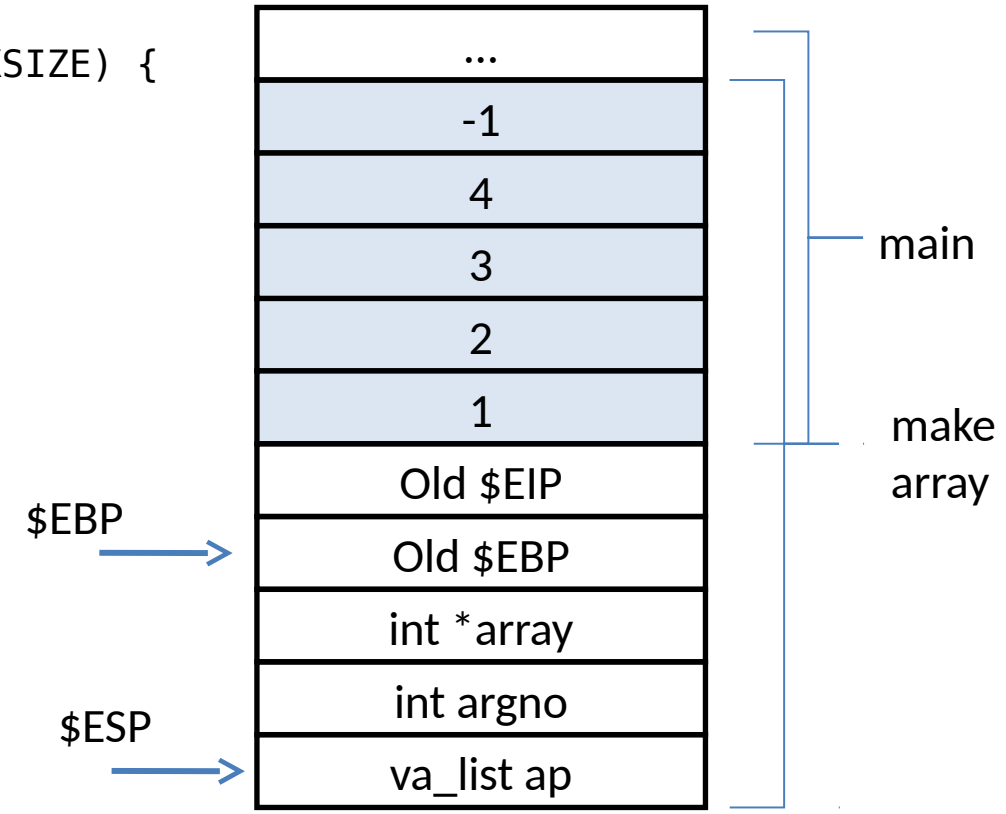
Proměnný počet parametrů - stdarg.h

```
int *makearray(int a, ...) {
   va_list ap;
   int *array = (int *)malloc(MAXSIZE * sizeof(int));
   int argno = 0;
   va_start(ap, a);
   while (a > 0 && argno < MAXSIZE) {
        array[argno++] = a;
        a = va_arg(ap, int);
   }
   array[argno] = -1;
   va_end(ap);
   return array;
}
```

va_list
va_start, va_arg,
va_end, va_copy

Volání

```
int *p;
int i;
p = makearray(1,2,3,4,-1);
for(i=0;i<5;i++)
  printf("%d\n", p[i]);
```



Proměnný počet argumentů, více iterací

```
int *makearray(int a, ...) {
        va_list ap, ap1;
        int tmp;
        size_t count = 1;
        int argno = 0;
        va_start(ap, a);
        va_copy(ap1, ap);
        tmp = a;
        while (tmp > 0) {
                tmp = va_arg(ap1, int);
                count++;
        }
        int *array = (int *)malloc(count * sizeof(int));
        while (argno < count) {
                array[argno++] = a;
                a = va_arg(ap, int);
        }
        array[argno] = -1;
        va_end(ap);
        return array;
}
```

```
#include <stdlib.h>
#include <stdarg.h>
```

va_list
va_start, va_arg,
va_end, va_copy

Funkce printf a vprintf

```
#include <stdio.h>
#include <stdarg.h>

int vprintf ( const char * format, va_list arg ) {
     for arguments in format switch
        case 'd':  int val_i = va_arg(arg, int); break;
        case 'f':  float val_f = va_arg(arg, float); break;
}

void printf( const char * format, ... ) {
    va_list args;
    va_start (args, format);
    vprintf (format, args);
    va_end (args);
}

int main () {
    printf ("Call with %d variable argument.\n",1);
    return 0;
}
```

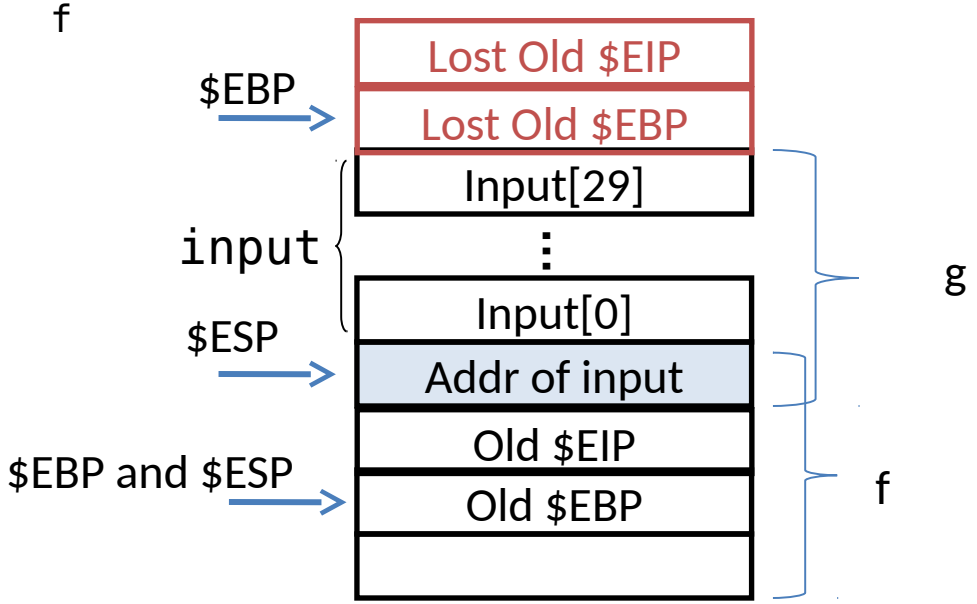
```
#include <stdlib.h>
#include <stdarg.h>
```

va_list
va_start, va_arg,
va_end, va_copy

Přetečení na zásobníku – buffer overflow

```
void f(char *s)
{
  gets(s);
}

int g()
{
  char input[30];
  f(input);
}
```

```
g:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $40, %esp
    andl    $-16, %esp
    subl    $16, %esp
    leal    -40(%ebp), %eax
    movl    %eax, (%esp)
    call    f
    leave
    ret
```



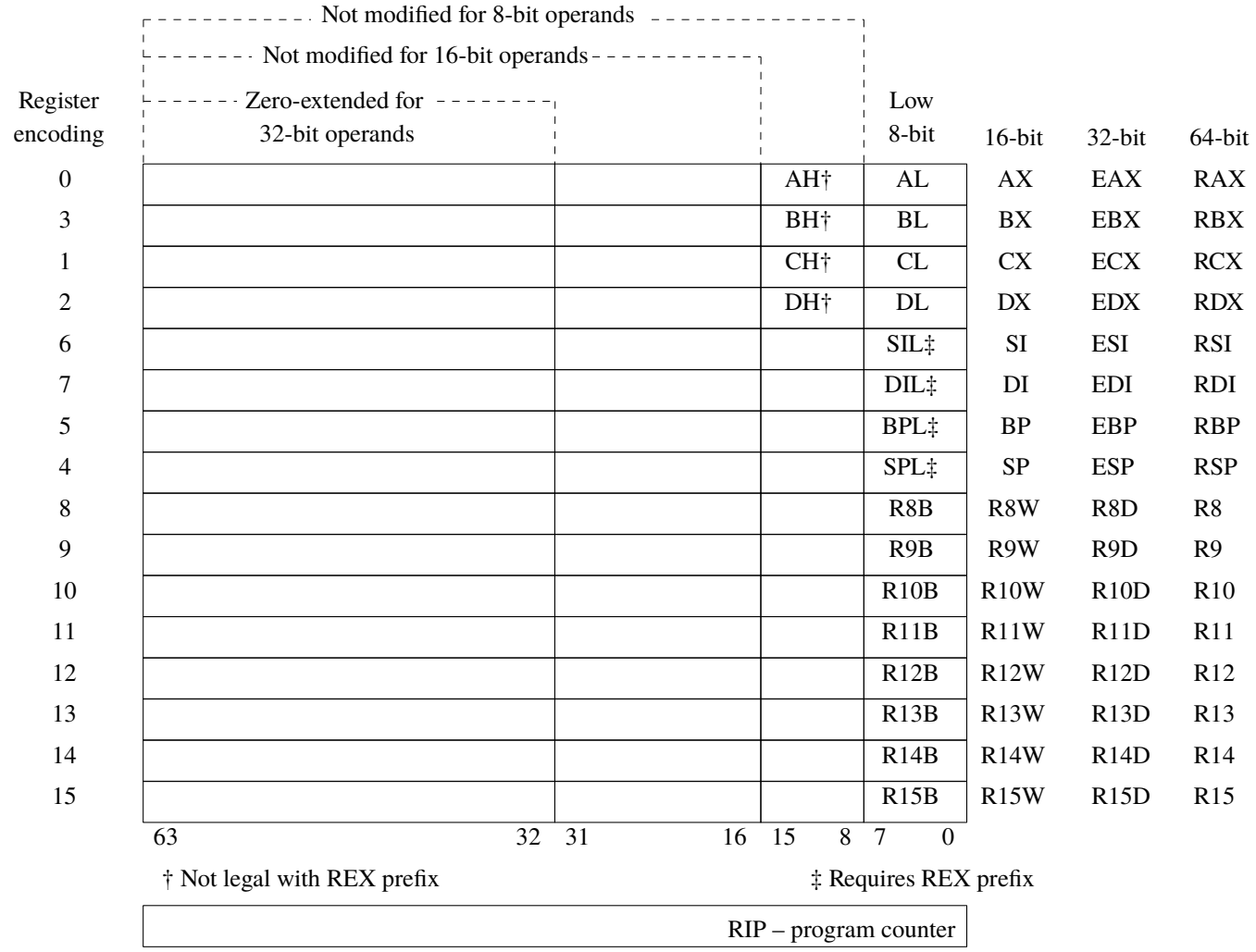
x86: Příklad s více argumenty

```
int simple(int a, int b, int c, int d, int e, int f){
  return a-f;
}
int main(){
  int x;
  x=simple(1, 2, 3, 4, 5, 6);
  return 0;
}
```

```
_simple:
pushl %ebp
movl  %esp, %ebp
movl  28(%ebp),%eax
movl  8(%ebp),%edx
movl  %edx, %ecx
subl  %eax, %ecx
movl  %ecx, %eax
popl  %ebp
ret
```

```
_main:
pushl %ebp            ebp saved on stack, push modifies esp
movl  %esp, %ebp      esp to ebp
andl  $-16, %esp      allign stack to 16-bytes
subl  $48, %esp       esp = esp – 48 (allocate space)
call  ___main
movl  $6, 20(%esp)    the last argument
movl  $5, 16(%esp)    the fifth argument
movl  $4, 12(%esp)    ...
movl  $3, 8(%esp)     ...
movl  $2, 4(%esp)     ...
movl  $1, 0(%esp)     the first argument
call  _simple         call the function
movl  %eax, 44(%esp)  store result to global x = simple(…);
movl  $0, %eax        return 0;
leave
ret
```

Volací konvence x86 pro 64-bit mód – registry



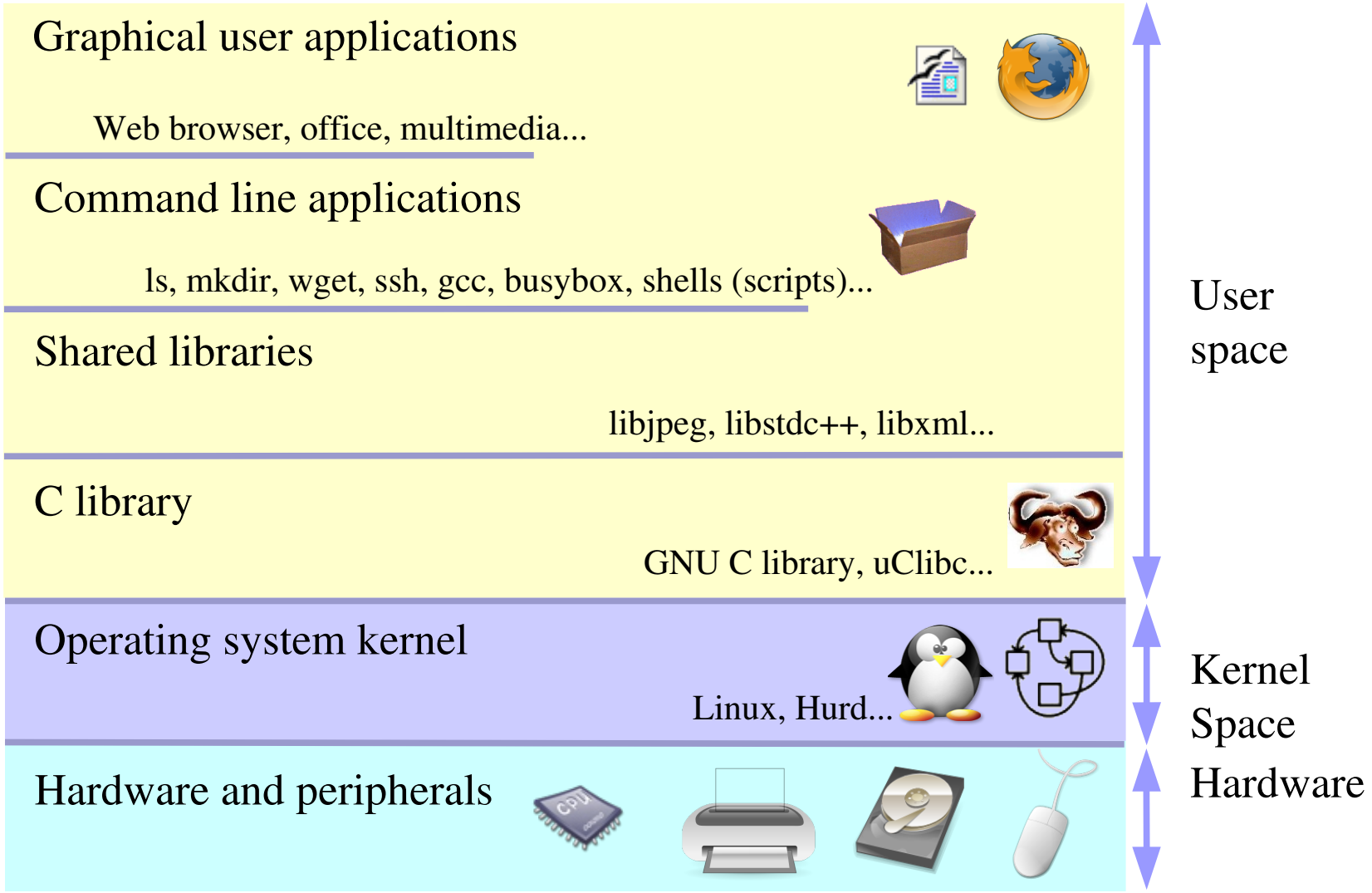
Source: Yasm manual https://www.tortall.net/projects/yasm/manual/manual.pdf

x86 v 64 bit režimu ()

System V Application Binary Interface
AMD64 Architecture Processor Supplement

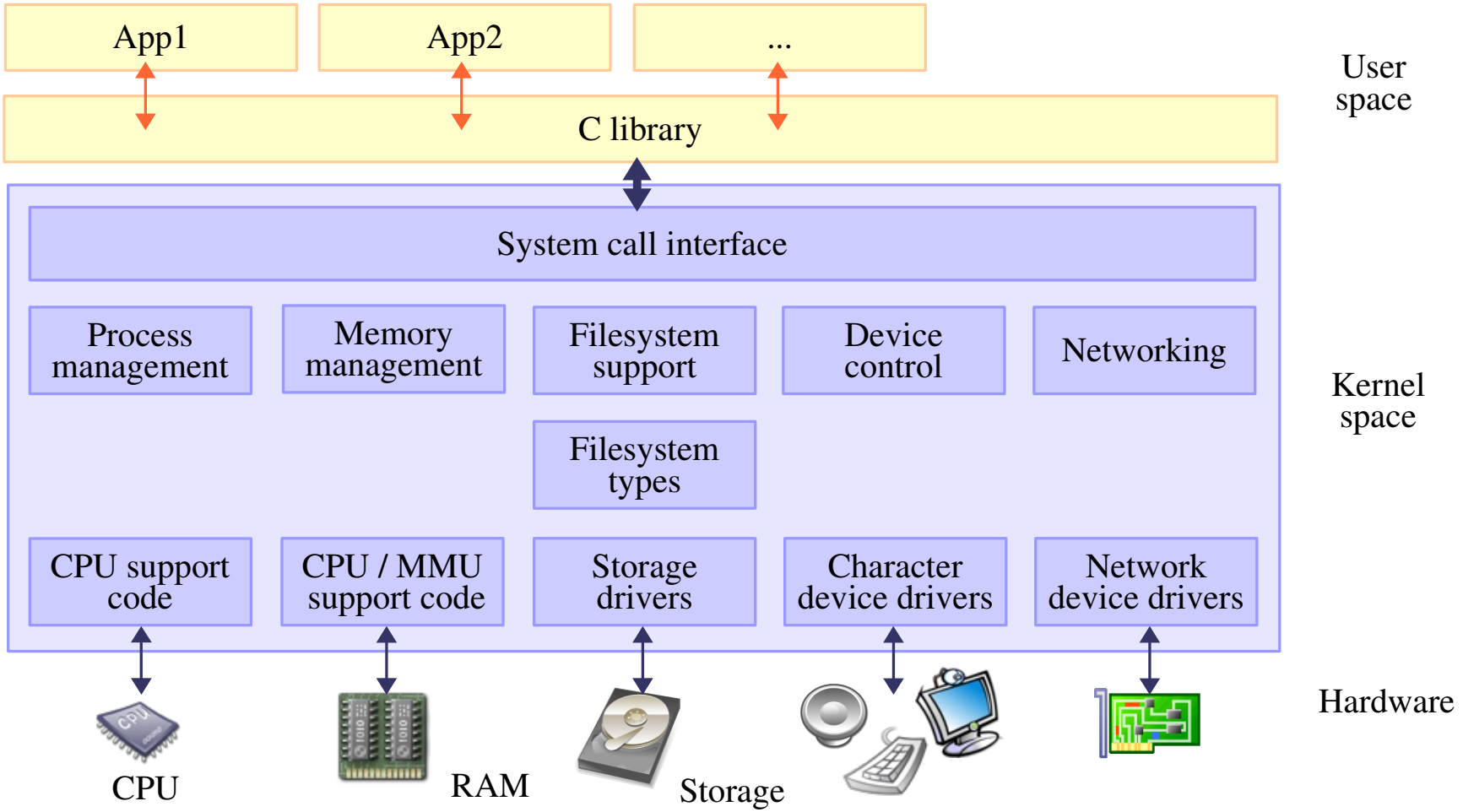
- Původní 32-bit volací konvence je nevýhodná, příliš mnoho přístupů na zásobník
- 64-bitové registry $RAX, $RBX, $RCX, $RDX, $RSI, $RDI, $RBP, $RSP, $R8 až R15 a množství multimediálních registrů
- Podle AMD64 konvence až 6 celočíselných parametrů v registrech $RDI, $RSI, $RDX, $RCX, $R8 a $R9
- Prvních 8 parametrů double a float v XMM0-7
- Návratová hodnota v $RAX
- Zásobník zarovnaný na 16 bytů
- Pokud není jistota, že funkce přijímá pouze pevný počet parametrů (bez va_arg/...) je uložen do $RAX počet parametrů předaných v SSE registrech (nutnost va_copy)

Operační systém – jednotlivé úrovně



Source: https://bootlin.com/ (Michael Opdenacker)

Bloková struktura operačního systému



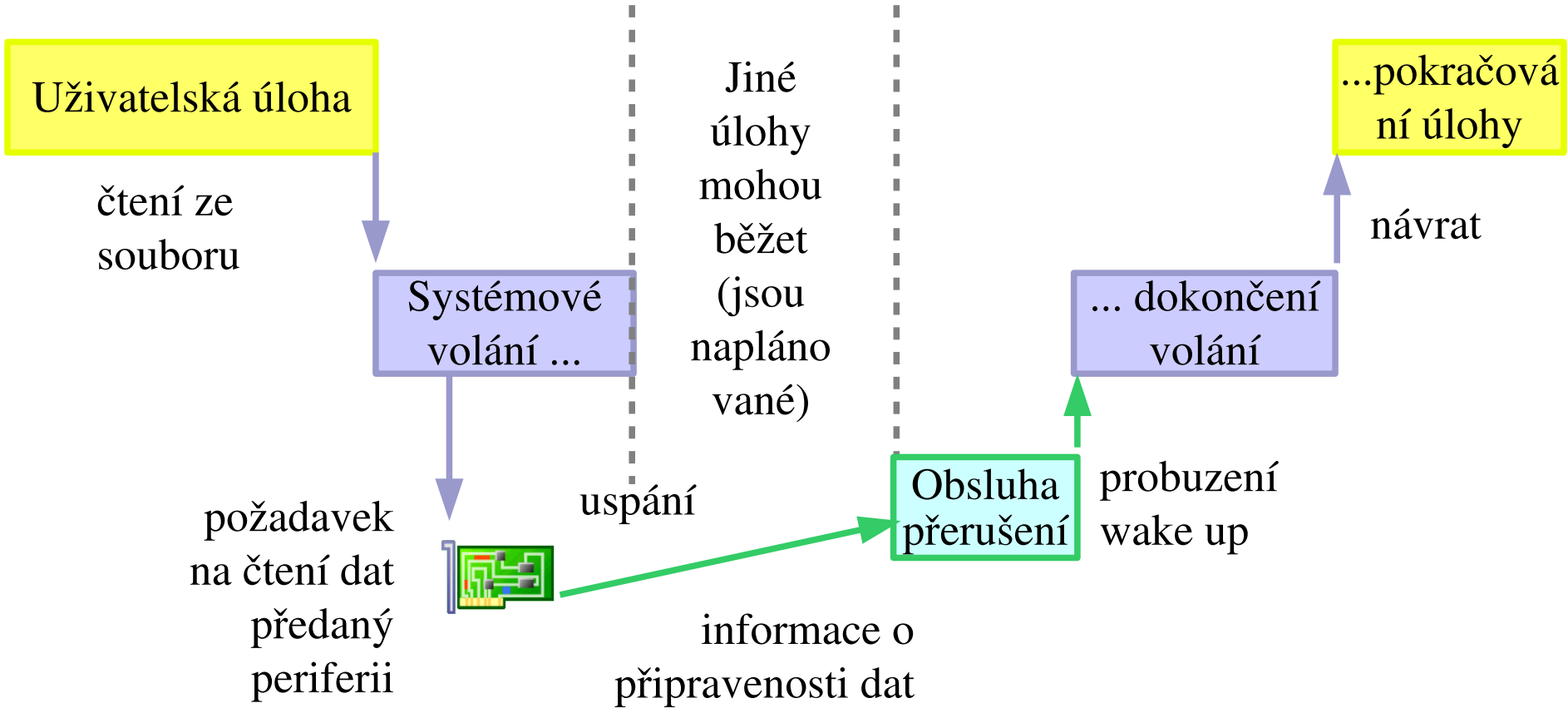
Source: https://bootlin.com/ (Michael Opdenacker)

Systémová volání

- Hlavní rozhraní mezi službami operačního systému a uživatelským prostorem/aplikacemi
- Na Linuxu, okolo 400 systémových volání služeb jádra
- Práce se soubory, zařízeními, síťové služby, komunikace mezi procesy, správa procesů, mapování paměti, časovače, vlákna, synchronizace, atd.
- Toto rozhraní se v čase nemění: volání mohou být programátory jádra pouze přidávaná
- Vlastní systémová volání jsou C knihovnou obsloužená/zabalená formou funkcí, uživatelské aplikace ubvykle žádná volání nevyužívají přímo ale pouze je volají přes odpovídající funkce C knihovny

Systémová volání – zopakování z přednášky o výjimkách

Systémové volání vyžaduje z důvodu ochrany paměti jádra OS přepnutí režimu procesoru z uživatelského módu do systémového.

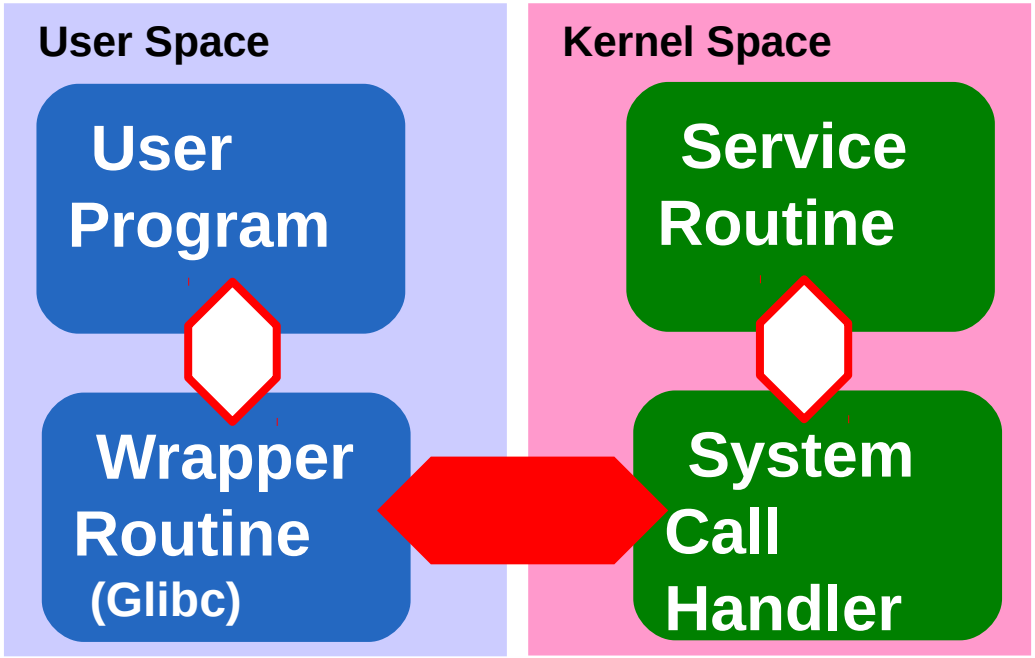


Source: https://bootlin.com/ (Michael Opdenacker)

Systémové volání - kroky

- Systémové služby (např. open, close, read, write, ioctl, mmap) jsou většinou běžným programům zpřístupněny přes běžné funkce C knihovny (GLIBC, CRT atd.), kterým se předávají parametry běžným způsobem
- Knihovní funkce pak přesune parametry nejčastěji do smluvených registrů, kde je očekává jádro OS
- Do zvoleného registru vloží číslo systémové služby (EAX na x86)
- Vyvolá výjimku (x86 Linux např int 0x80 nebo sysenter)
- Obslužná rutina jádra podle čísla dekóduje parametry služby a zavolá již obvyklým způsobem výkonnou funkci
- Jádro uloží do registru návratový kód a provede přepnutí zpět do uživatelského režimu
- Zde je vyhodnoceno hlášení chyb (errno) a běžný návrat do volajícího programu

Volání jádra – wrapper a obslužná rutina



Parametry některých systémových volání (Linux i386)

%eax	Name	Source	%ebx	%ecx	%edx	%esx	%edi
1	sys_exit	kernel/exit.c	int				
3	sys_read	fs/read_write.c	unsigned int	char *	size_t		
4	sys_write	fs/read_write.c	unsigned int	const char *	size_t		
5	sys_open	fs/open.c	const char *	int	mode_t		
6	sys_close	fs/open.c	int				
15	sys_chmod	fs/open.c	const char	mode_t			
20	sys_getpid	kernel/timer.c	void				
21	sys_mount	fs/namespace.c	char *	char *	char *	unsigned long	void *
88	sys_reboot	kernel/sys.c	int	int	unsigned int	void *	

System Call Number

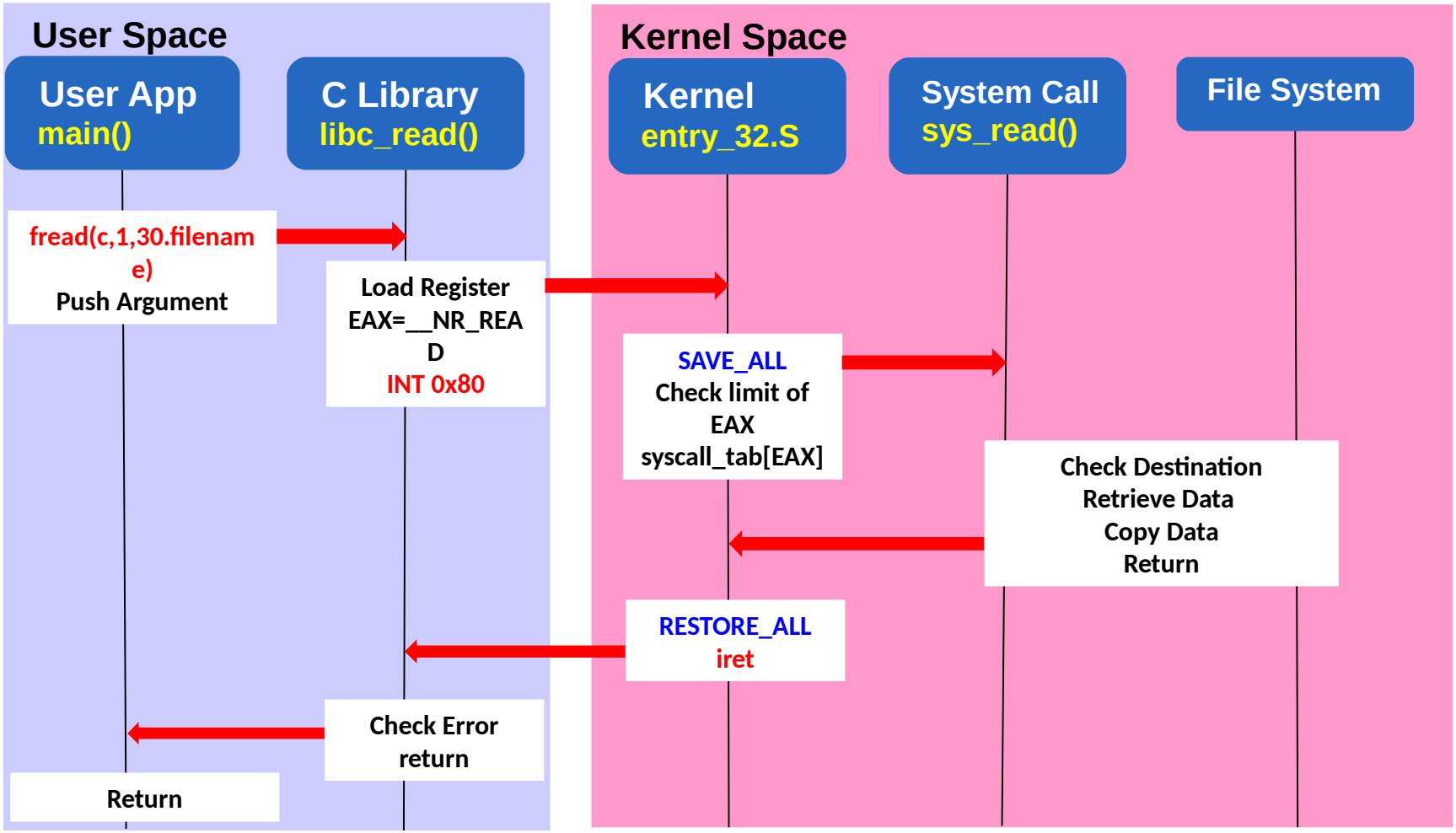
System Call Name

First parameter

Second parameter

Third parameter

Průběh volání



Systémového volání pro architekturu MIPS

Register	use on input	use on output	Note
$at	—	(caller saved)	
$v0	syscall number	return value	
$v1	—	2nd fd only for pipe(2)	
$a0 ... $a2	syscall arguments	returned unmodified	O32
$a0 ... $a2, $a4 ... $a7	syscall arguments	returned unmodified	N32 and 64
$a3	4th syscall argument	$a3 set to 0/1 for success/error	
$t0 ... $t9	—	(caller saved)	
$s0 ... $s8	—	(callee saved)	
$hi, $lo	—	(caller saved)	

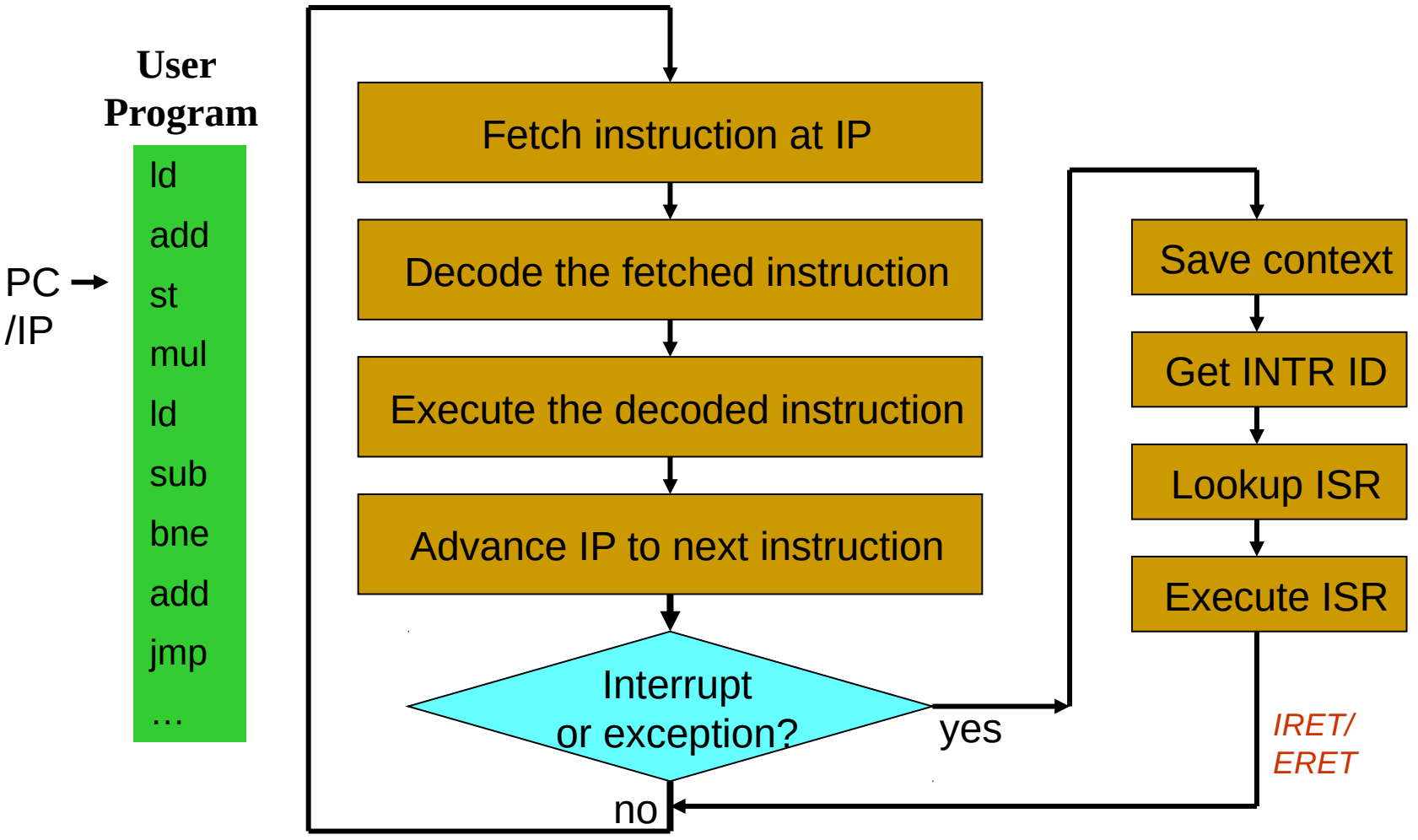
Vlastní systémové volání je reprezentované instrukcí **SYSCALL**, přiřazení čísel **http://lxr.linux.no/#linux+v3.8.8/arch/mips/include/uapi/asm/unistd.h**

Zdroj: http://www.linux-mips.org/wiki/Syscall

Hello World – první MIPS program na Linuxu

```
#include <asm/unistd.h>
#include <asm/asm.h>
#include <sys/syscall.h>

#define O_RDWR              02
        .set    noreorder
        LEAF(main)
#       fd = open("/dev/tty1", O_RDWR, 0);
        la      a0,tty
        li      a1,O_RDWR
        li      a2,0
        li      v0,SYS_open
        syscall
        bnez    a3,quit
        move    s0,v0                   # delay slot
#       write(fd, "hello, world.\n", 14);
        move    a0,s0
        la      a1,hello
        li      a2,14
        li      v0,SYS_write
        syscall
#       close(fd);
        move    a0,s0
        li      v0,SYS_close
        syscall

quit:
        li      a0,0
        li      v0,SYS_exit
        syscall

        j       quit
        nop

        END(main)

        .data
tty:    .asciz  "/dev/tty1"
hello:  .ascii  "Hello, world.\n"
```

Přerušení a výjimky ve výkonném cyklu počítače



Exceptions sources on MIPS32

- Exceptions caused by hardware malfunctioning:
  - **Machine Check**: Processor detects internal inconsistency;
  - **Bus Error**: on a load or store instruction, or instruction fetch;
- Exceptions caused by some external causes (to the processor):
  - **Reset**: A signal asserted on the appropriate pin;
  - **NMI**: A rising edge of NMI signal asserted on an appropriate pin;
  - **Hardware Interrupts**: Six hardware interrupt requests can be made via asserting signal on any of 6 external pins.

    Hardware interrupts can be masked by setting appropriate bits in Status register;

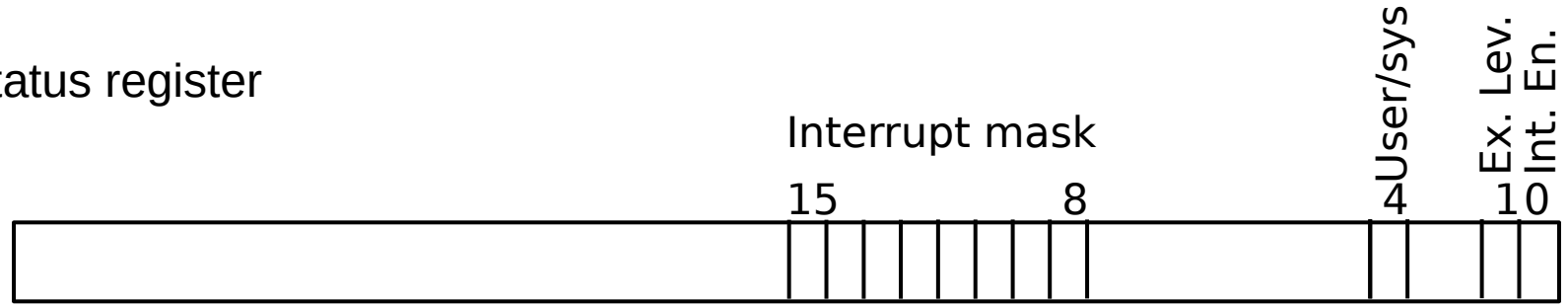
Exceptions sources on MIPS32 - continued

- Exceptions that occur as result of instruction problems:
  - **Address Error**: a reference to a nonexistent memory segment, or a reference to Kernel address space from User Mode;
  - **Reserved Instruction**: A undefined opcode field (or privileged instruction in User mode) is executed;
  - **Integer Overflow**: An integer instruction results in a 2's complement overflow;
  - **Floating Point Error**: FPU signals one of its exceptions, e.g. divide by zero, overflow, and underflow)
- Exceptions caused by executions of special instructions:
  - **Syscall**: A Syscall instruction executed;
  - **Break**: A Break instruction executed;
  - **Trap**: A condition tested by a trap instruction is true;

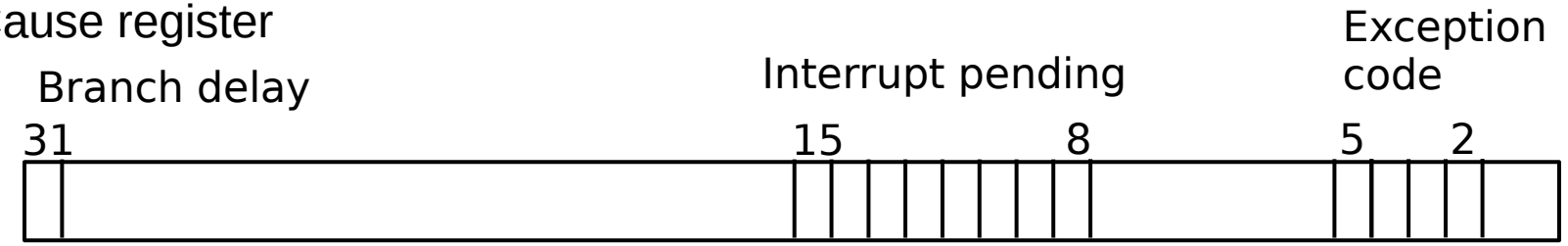
MIPS – registers for exceptions status and control

Register name	Register number	Usage
Status	12	Interrupt mask and enable bits
Cause	13	Exception type
EPC	14	Following address of the instruction the exception occurred or address of jump/branch instruction when exception in delay slot

Status register



Cause register



MIPS – codes of the exception sources

Number	Name	Cause of exception
0	Int	interrupt (hardware)
4	AdEL	address error exception (load)
5	AdES	address error exception (store)
6	IBE	bus error on instruction fetch
7	DBE	bus error on data load or store
8	Sys	syscall exception
9	Bp	breakpoint exception
10	RI	reserved instruction execution
11	CpU	coprocessor unimplemented
12	Ov	arithmetics overflow exception
13	Tr	trap
15	FPE	floating point

MIPS – exception/interrupt processing

CPU accepts interrupt request, exception or **syscall** opcode

```
EPC <= PC
Cause <= (cause code for event)
Status <= Status | 2
PC <= (handler address)
```

Interrupt service routine/exception handler startup is responsible for
- identification of request cause from co-processor 0 **mfc0 rd, rt**
- CPU state can be controlled by instruction **mtc0 rd, rt**
- **rd** is gen. purpose register, **rt** is one of co-processor 0 registers

The **eret** instruction finalizes exception handling and enables exceptions

```
PC <= EPC
Status <= Status & ~2
```

Precise exception processing

- If interrupt/exception is successfully handled (i.e. missing page has been swapped in, etc.) and execution continues at instruction before which interrupt has been accepted, then interrupted code flow is not altered and cannot detect interruption (except for delay/timing and cases when state modification is intended/caused by system call)
- Remark: Precise exception handling is most complicated by delayed writes (and superscalar CPU instruction reordering) which leads to synchronous exceptions detected even many instruction later than causing instruction finishes execution phase. Concept of state rewind or “transactions” confirmation is required for memory paging in such systems.

Evaluation of the exception source

- Software cause evaluation (polled exception handling)
  - All exceptions/interrupts start same routine at same address – i.e. for MIPS that routine starts at EBase + 0x180 (default 0x80000180).
  - Routine reads source from status register (MIPS: cause registr)
- Vectored exception handling
  - CPU support hardware identifies cause/source/interrupt number
  - Array of ISR start addresses is prepared on fixed or preset (VBR – vector base register) address in main memory
  - CPU computes index into table based on source number
  - CPU loads word from given address to PC
- Non-vectored exception handling with more routines/initial addresses assigned to exception classes and IRQ priorities
- Additional combinations when more addresses are used for some division into classes or some helper HW provides decoding speedup

Asynchronous and synchronous exceptions/interrupts

- External interrupts/exceptions are generally asynchronous – i.e. they are not tied to some instruction
  - RESET- CPU state initialization and (re)start form initial address
  - NMI - non-maskable interrupt (temperature/bus/EEC fault)
  - INT - maskable/regular interrupts (peripherals etc.)
- Synchronous exceptions (and or interrupts) are result of exact instruction execution
  - Arithmetic overflow, division by zero etc.
  - TRAP - debugger breakpoint, exception after each executed instruction for single-stepping, etc.
  - Modification of interrupted code flow state (registers, flags, etc.) is expected for some of these causes (unknown instruction emulation, system calls, jump according to program provided exception tables, etc.)