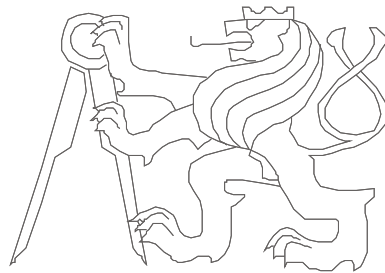


# Computer Architectures

Parameters Passing to Subroutines and Operating System  
Implemented Virtual Instructions (System Calls)



Czech Technical University in Prague, Faculty of Electrical Engineering

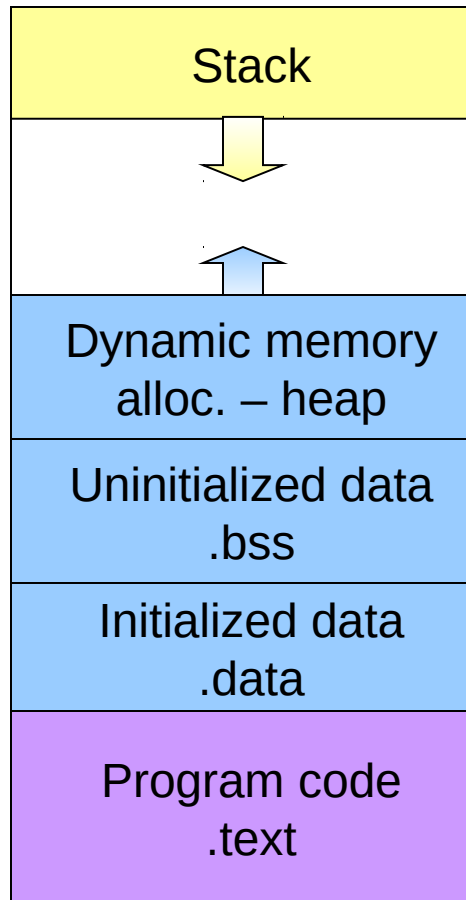
# Kinds of calling conventions and parameters passing

- Regular (standard) functions (subroutines) calling
  - Parameters passing – in registers, on stack, through register windows
  - Calling convention (part of ABI – application binary interface) selected according to the CPU architecture supported options, agreement between compilers, system and additional libraries is required

(x86 Babylonian confusion of tongues - cdecl, syscall, optlink, pascal, register, stdcall, fastcall, safecall, thiscall MS/others)
  - Stack frames allocated space for local variables and C-library `alloca()`
- System calls (requests a service from OS kernel)
  - Control transfer (switching) from user to system (privileged) mode
- Remote functions calls and method invocations
  - Callee cannot read (easily) from memory space of caller process
  - Standards for network cases (RPC, CORBA, SOAP, XML-RPC)
  - Machine local same as networked + more: OLE, UNO, D-bus, etc.

# Program layout in memory at process startup

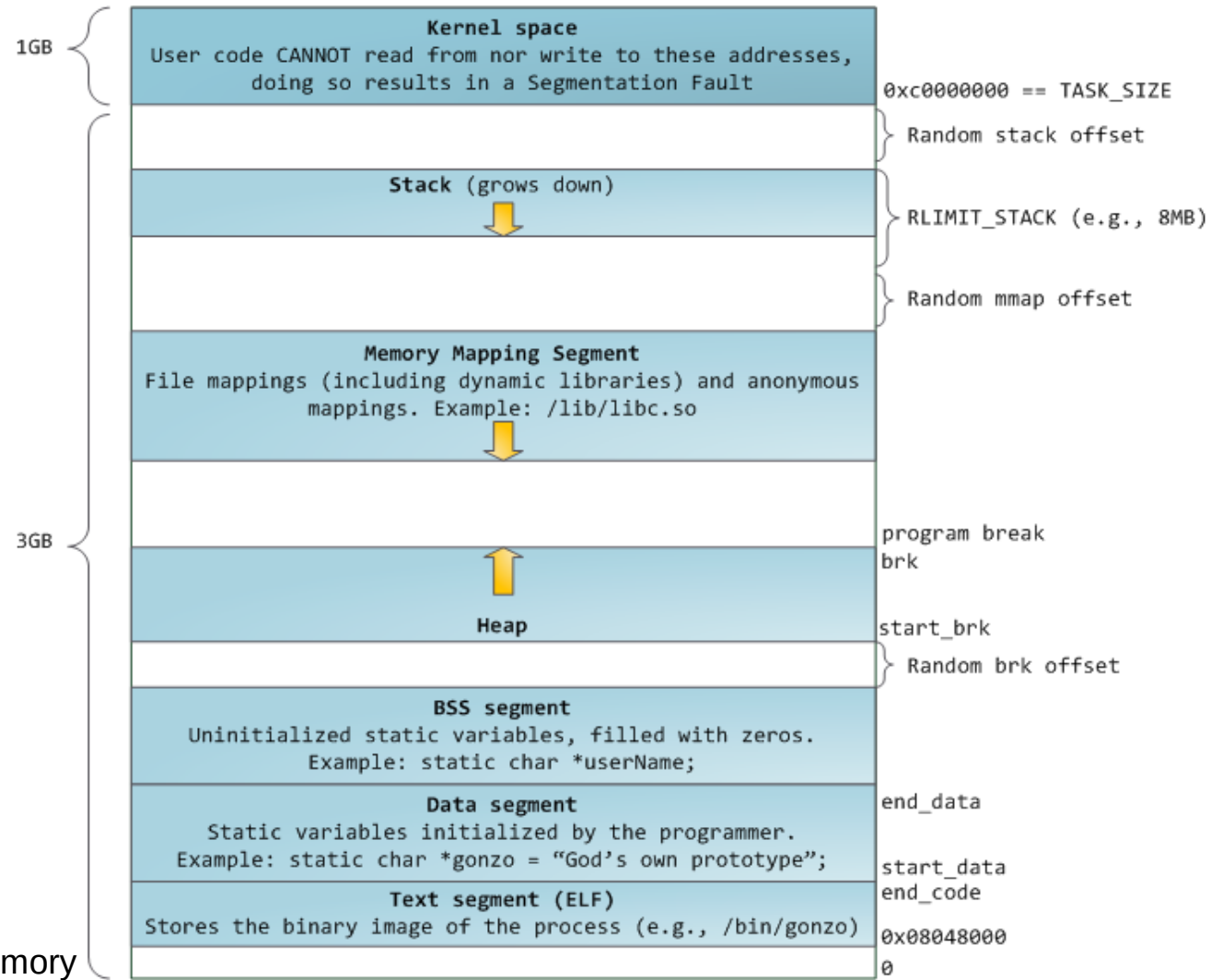
0x7fffffff



0x00000000

- The executable file is mapped (“loaded”) to process address space – sections **.data** and **.text** (note: LMA != VMA for some special cases)
- Uninitialized data area (**.bss** – block starting by symbol) is reserved and zeroed for C programs
- Stack pointer is set and control is passed to the function **\_start**
- Dynamic memory is usually allocated above **\_end** symbol pointing after **.bss**

# Process address space (32-bit x86 Linux)



Based on:  
 Anatomy of a Program in Memory  
<http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory>

# Steps processed during subroutine call

- Calling routine/program (caller) calculates the values of the parameters
- Caller prepares for overwrite of registers which ABI specifies as clobberable (can be modified by callee) storing values which are still needed
- Parameters values are placed in registers and or on stack according to used ABI calling convention
- Control is transferred to the first subroutine instruction while return address is saved on stack or stored in dedicated register
- Subroutine saves previous values of registers, which are to be used and are specified as callee saved/non-clobberable
- Space for local variables is allocated on stack
- The body of subroutine is executed
- The function result is placed to register(s) defined by calling convention
- Values of callee saved/non-clobberable registers are restored
- Return from subroutine instruction is executed, it can release stack space used for parameters but it is usually caller duty to adjust stack to free parameters

## Example: MIPS calling convention and registers usage

- a0 – a3: arguments (registers \$4 – \$7)
- v0, v1: registers to hold function result value (\$2 and \$3)
- t0 – t9: temporary/clobberable registers (\$8-\$15,\$24,\$25)
  - callee function can use them as it needs without saving
- at: temporary register for complex assembly constructs (\$1)
- k0, k1: reserved for operating system kernel (\$26, \$27)
- s0 – s7: saved (non-clobberable) registers (\$16-\$23)
  - if used by callee, they has to be saved first and restored at return
- gp: global static data pointer (\$28)
- sp: stack pointer (\$29) – top of the stack, grows down to 0
- fp: frame pointer (\$30) – points to start of local variables on stack
- ra: return address register (\$31) – implicitly written by **jal** instruction – jump and link – call subroutine

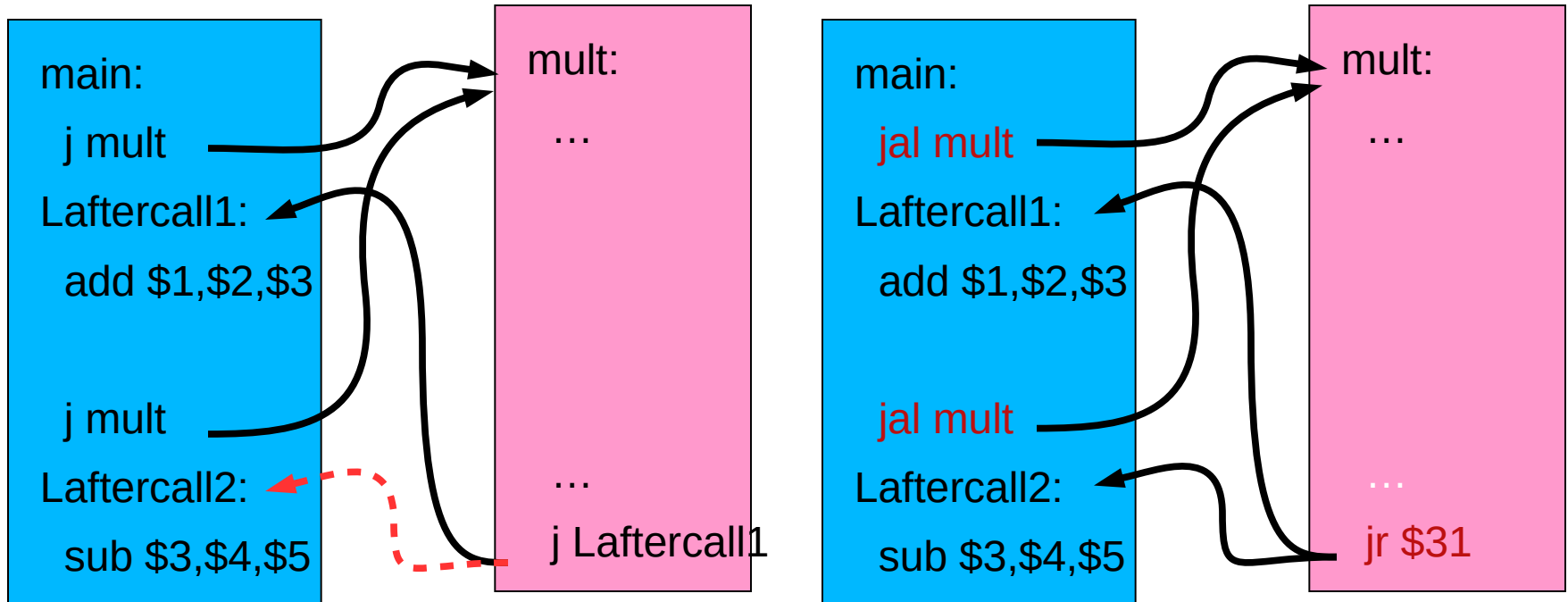
## MIPS: call instruction and return

- Subroutine invocation (calling): jump and link
  - **jal** ProcedureLabel
  - Address of the second (consider delay slot) instruction following **jal** is stored to **ra** (\$31) register
  - Target (subroutine start) address is filled to **pc**
- Return from register: jump register
  - **jr ra**
  - Loads **ra** register content to **pc**
  - The same instruction is used when jump target address is computed or chosen from table – i.e. case/switch statements in C source code

# Jump and subroutine call differences

Jump does not save return address

⇒ cannot be used to call subroutine from more locations



Example author: Prof. Sirer  
Cornell University

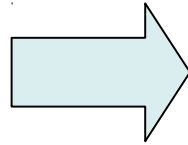
on the contrary, call using register `ra`  
allows you to call a subroutine from  
as many locations as needed



# Jump and Link Instruction in Detail

C/C++

```
int main() {  
    simple();  
    ...  
}  
  
void simple(){  
    return;  
}
```



MIPS

```
0x00400200 main: jal simple  
0x00400204 ...  
  
0x00401020 simple: jr $ra
```

Description:

For procedure call.

Operation:

$\$31 = PC + 8; PC = ((PC+4) \& 0xf0000000) | (target \ll 2)$

Syntax:

jal target

Encoding:

0000 11ii iiiiiiii iiiiiiii iiiiiiii iiiiiiii

Caller/**jal** stores return address into **ra** (reg \$31) and transfers control to the callee/subroutine first instruction. Call returns by jump to return address (**jr \$ra**).

## Code example

C language source code:

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

Parameters **g**, ..., **j** are placed in registers **a0**, ..., **a3**

**f** function result is computed in **s0** (**s0** non-clobberable and previous value has to be saved on stack)

Function return value is expected in **v0** register by caller

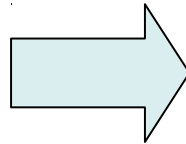
# MIPS: Caller and Callee with Parameters Passing

- \$a0-\$a3 arguments values
- \$v0-\$v1 function return value(s)

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
```

Clobbers  
caller  
guaranteed  
to be saved  
register \$s0

## Example code compiled for MIPS architecture

```
int leaf_fun (int g, h, i, j)
```

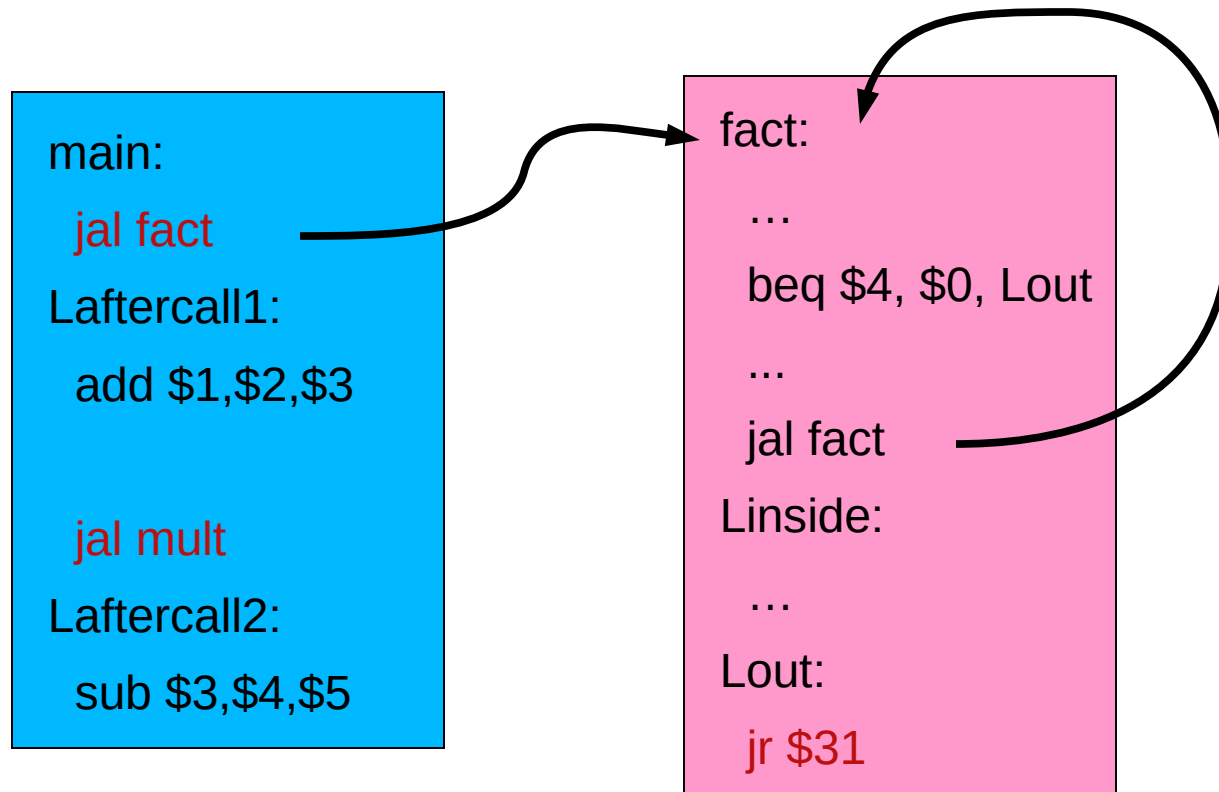
$g \rightarrow \$a0$ ,  $h \rightarrow \$a1$ ,  $i \rightarrow \$a2$ ,  $j \rightarrow \$a3$ ,  $\$v0$  – ret. val,  $\$sX$  – save,  $\$tX$  – temp,  $\$ra$  – ret. addr

leaf_fun:	
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

Listing before filling/moving instructions into delay-slots

Source: Henesson: Computer Organization and Design

# Link-register and nested or recursive calling problem



The value of **ra** register has to be saved before another (nested or recursive) call same as for non-clobberable (saved) registers **sX**

## Recursive function or function calling another one

C language source code:

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

Function parameter **n** is located and passed in **a0** register

Function return value in **v0** register

# MIPS: Recursive Function Example

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

Listing before filling/moving instructions into delay-slots

# MIPS Calling Convention and ABI

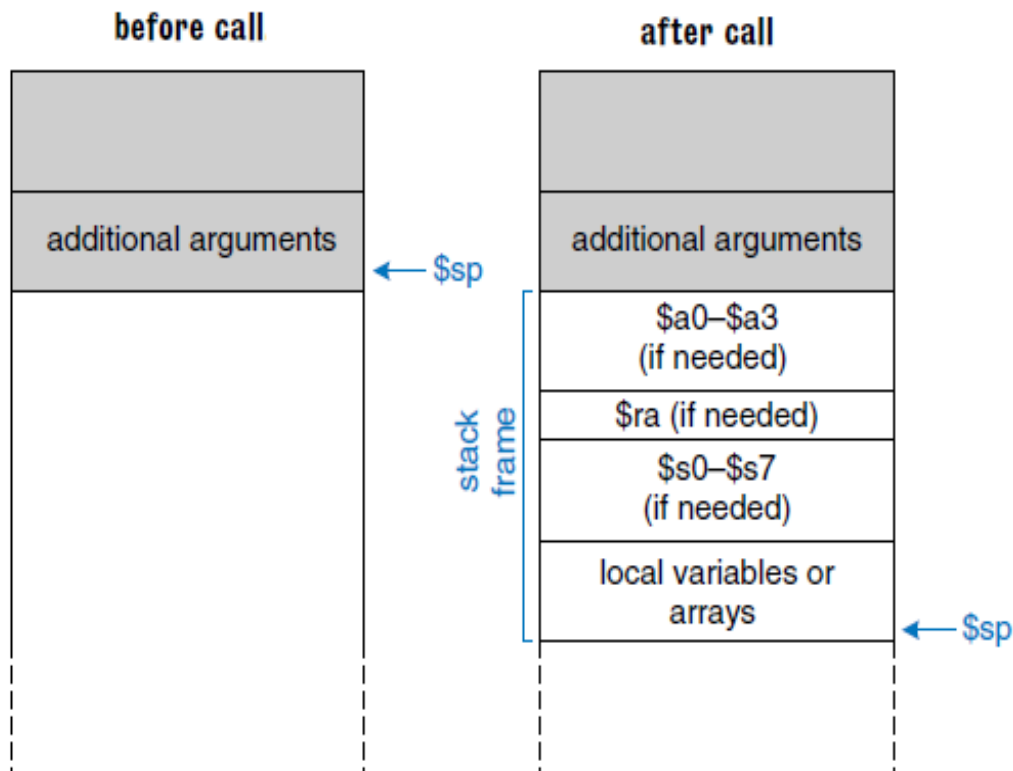
- Which registers needs to be saved by callee is defined by Application binary interface (ABI) for given architecture

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: Subroutine with More than 4 Arguments

- MIPS calling convention: The first four in  $\$a0 \dots \$a3$ , other placed on stack such way that fifth argument is found directly at  $\$sp$  and next at  $\$sp+4$ . Space is allocated and deallocated by caller.



```
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
```

# C calling convention for x86 – 32-bit mode registers

General-Purpose Registers			16-bit	32-bit	
	31	16 15	8 7	0	
Accumulator		AH	AL	AX	EAX
Base		BH	BL	BX	EBX
Count		CH	CL	CX	ECX
Data		DH	DL	DX	EDX
Source Index		SI			ESI
Destination Index		DI			EDI
Base Pointer		BP			EBP
Stack Pointer		SP			ESP

31		0			
	Program Status and Control Register		EFLAGS		
31		0			
	Instruction Pointer		EIP		
		15	Segment Registers	0	
			Code Segment		CS
			Data Segment		DS
			Stack Segment		SS
			Extra Segment		ES
					FS
					GS

## Standard C calling convention for x86 in 32-bit mode

- Registers \$EAX, \$ECX and \$EDX are clobberable/can be modified by subroutine without saving
- Registers \$ESI, \$EDI, \$EBX values has to be preserved
- Registers \$EBP, \$ESP have predefined use, sometimes even \$EBX use can be special (dedicated for GOT access)
- Three registers are usually not sufficient even for local variables of leaf-node functions
- Function result is expected in \$EAX register; for 64-bit return values \$EDX holds MSB part of the result
- Everything other – all parameters, local variables atc. have to be placed on stack

# x86: calling function with one parameter

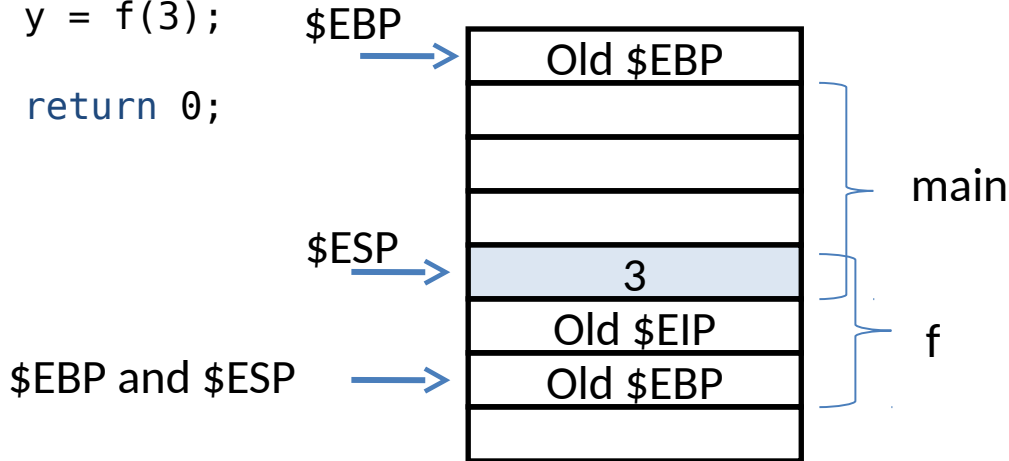
```
#include <stdio.h>
```

```
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: calling function with two parameters

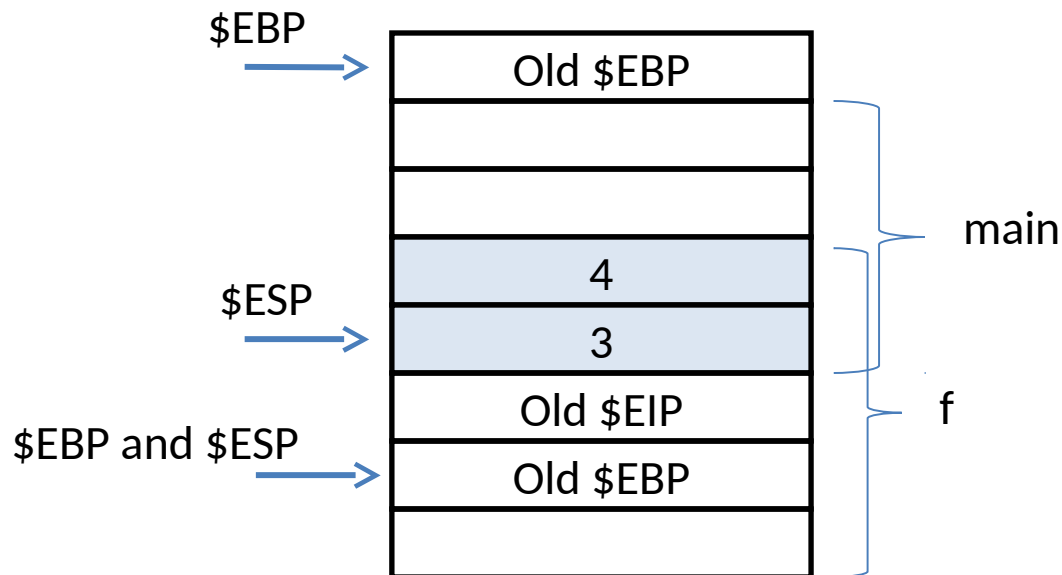
```
#include <stdio.h>

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
```



# Variable parameters count - 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;
}
```

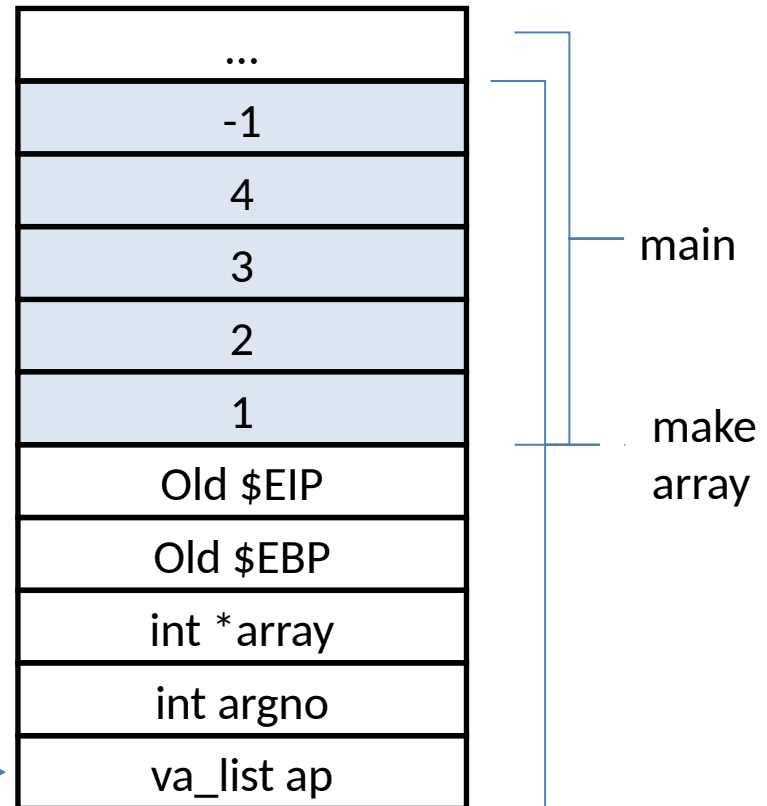
va\_list  
va\_start, va\_arg,  
va\_end, va\_copy

Volání

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

\$EBP →

\$ESP →



## Variable parameters count, multiple iterations

```
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
```

# Functions printf and 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
```

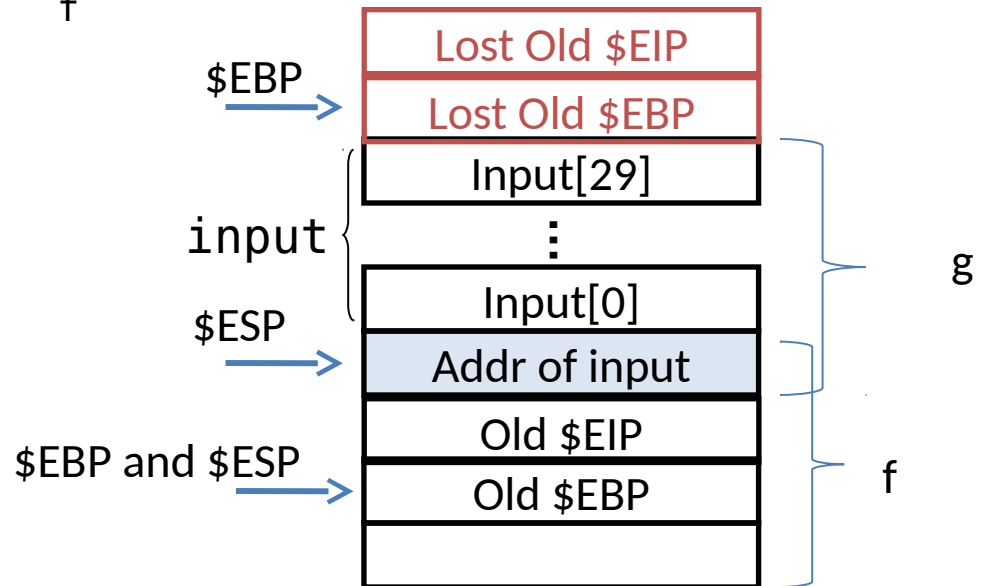


# Stack/buffer overflow error example

```
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
```

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

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



## x86: Example with More Arguments

```
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;
}
```

```

        _main:
        pushl %ebp                ebp saved on stack, push modifies esp
        movl  %esp, %ebp          esp to ebp
        andl  $-16, %esp          align stack to 16-bytes
        subl  $48, %esp           esp = esp - 48 (allocate space)
__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
        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
```

# x86 calling convention for 64-bit mode – registers

Register encoding	Zero-extended for 32-bit operands	Not modified for 16-bit operands	Not modified for 8-bit operands	Low 8-bit	16-bit	32-bit	64-bit
0			AH†	AL	AX	EAX	RAX
3			BH†	BL	BX	EBX	RBX
1			CH†	CL	CX	ECX	RCX
2			DH†	DL	DX	EDX	RDX
6				SIL‡	SI	ESI	RSI
7				DIL‡	DI	EDI	RDI
5				BPL‡	BP	EBP	RBP
4				SPL‡	SP	ESP	RSP
8				R8B	R8W	R8D	R8
9				R9B	R9W	R9D	R9
10				R10B	R10W	R10D	R10
11				R11B	R11W	R11D	R11
12				R12B	R12W	R12D	R12
13				R13B	R13W	R13D	R13
14				R14B	R14W	R14D	R14
15				R15B	R15W	R15D	R15

63
32 31
16 15 8 7
0

† Not legal with REX prefix ‡ Requires REX prefix

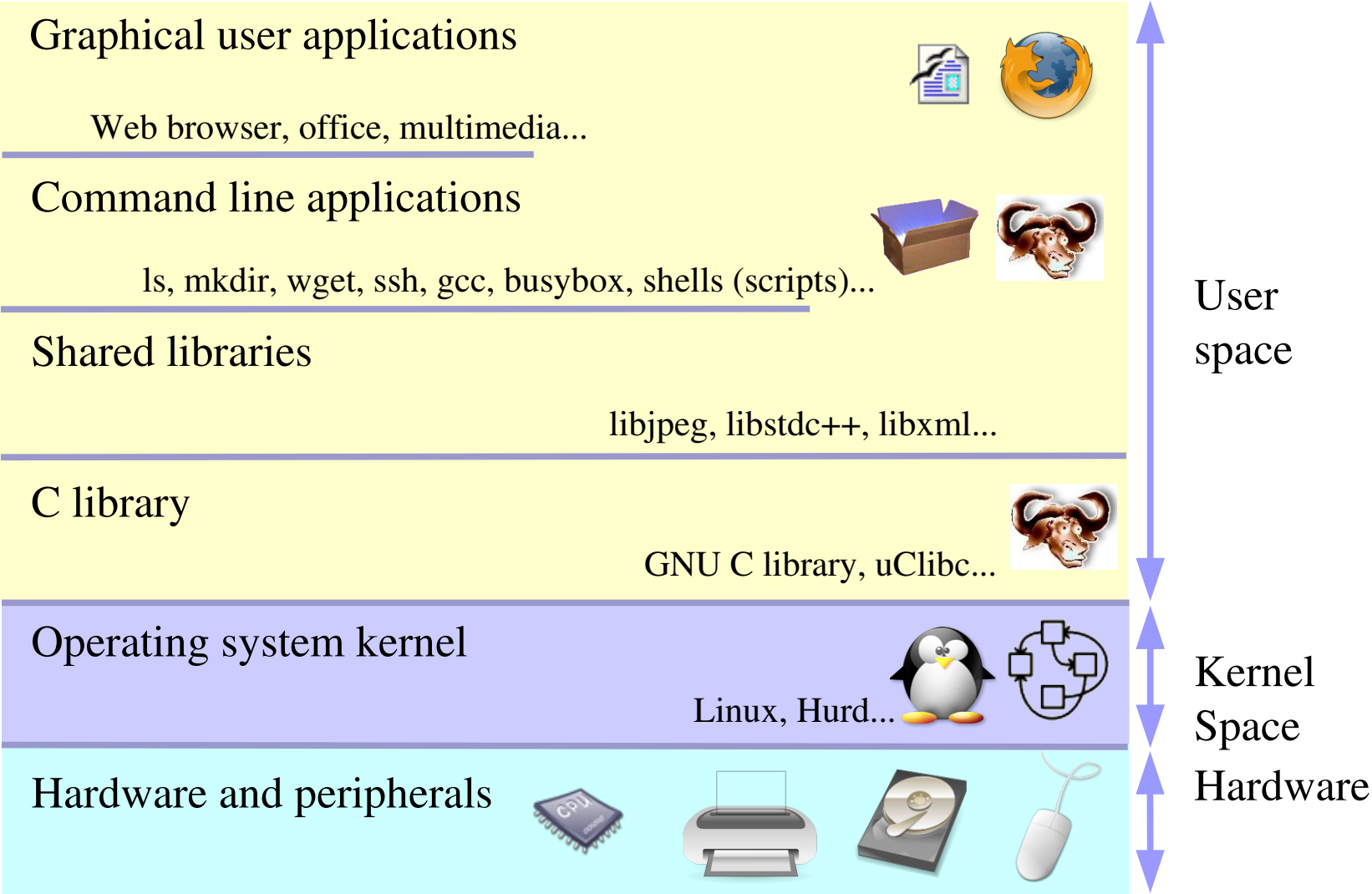
RIP – program counter

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

## x86 calling convention for 64-bit mode (for Linux)

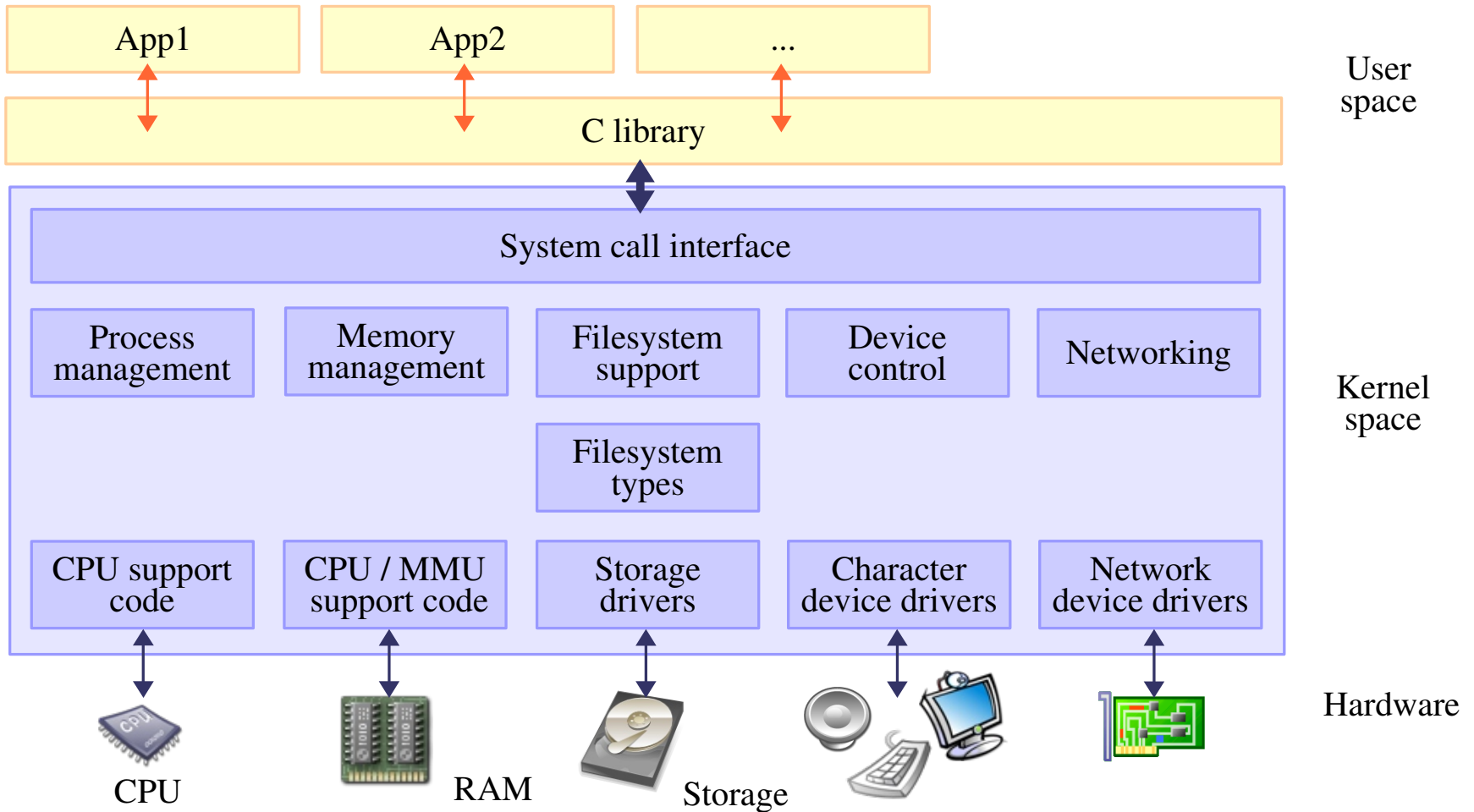
- Original 32-bit calling convention is expensive, too many main memory (stack) accesses
- 64-bit registers \$RAX, \$RBX, \$RCX, \$RDX, \$RSI, \$RDI, \$RBP, \$RSP, \$R8 ... R15 and many multimedia registers
- AMD64 ABI/calling convention places up to the first 6 integral data type parameters in \$RDI, \$RSI, \$RDX, \$RCX, \$R8 and \$R9 registers
- The first 8 double and float data type parameters in XMM0-7
- Return/function result value in \$RAX
- Stack is always 16-byte aligned when a call instruction is executed
- If calling function without prototype or function with variable arguments count (`va_arg/...`) then \$RAX has to be set to number of parameters passed in registers (never use `va_arg` twice without `va_copy`)

# Operating system – layers



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

# Operating system structure

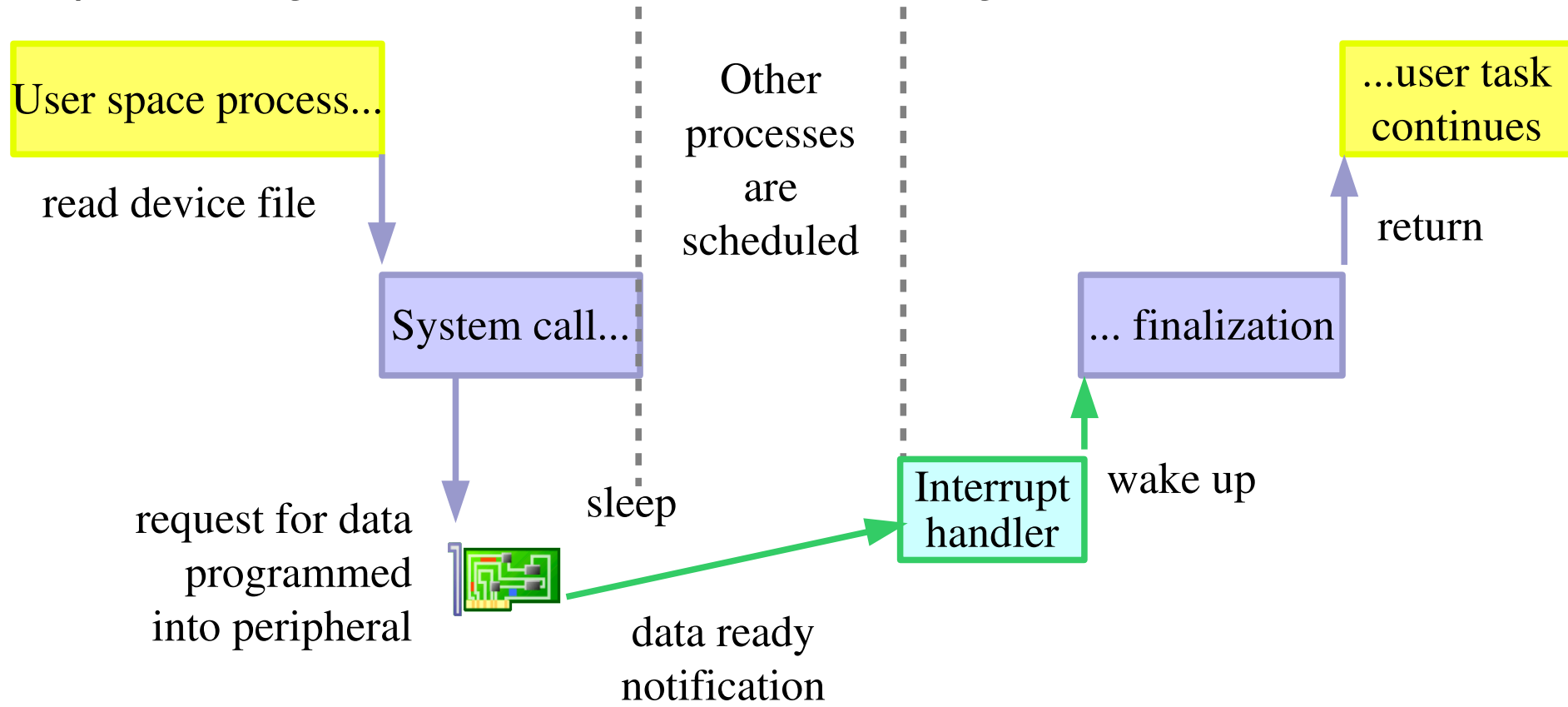


## System calls

- The main interface between the operating system kernel and user space is the set of system calls
- On Linux, about 400 system calls that provide the main kernel services
- File and device operations, networking operations, inter-process communication, process management, memory mapping, timers, threads, synchronization primitives, etc.
- This interface is stable over time: only new system calls can be added by the kernel developers
- This system call interface is wrapped by the C library, and user space applications usually never make a system call directly but rather use the corresponding C library function

# System call

When peripheral transfers data, task is suspended/waiting (and other work could be done by CPU). Data arrival results in IRQ processing, CPU finalizes transfer and original task continues

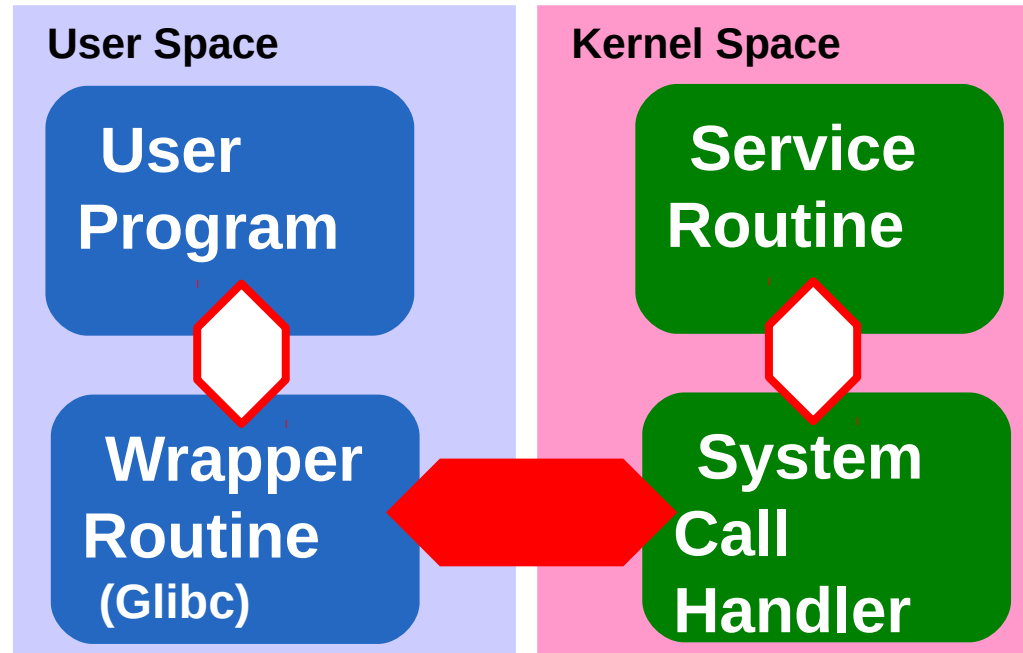




## System call processing steps

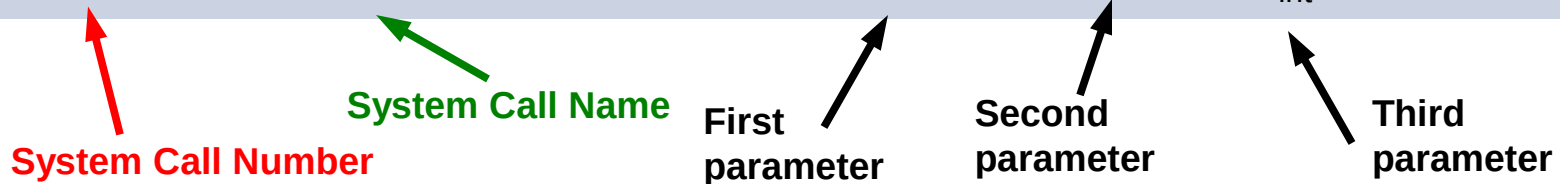
- Operating system services (i.e. open, close, read, write, ioctl, mmap) are wrapped by common runtime libraries (GLIBC, CRT atd.) and parameters are passed to wrappers same way as to the usual functions/subroutines
- Library in the most cases moves parameters values into registers specified by given system call ABI (differs from function calls ABI)
- Service identification / syscall number is placed in defined register (EAX for x86 architecture)
- Syscall exception/interrupt instruction is executed (int 0x80 or sysenter for x86 Linux)
- Syscall entry handler decodes parameters according to syscall number and calls system service function usual function ABI way
- Kernel return code is placed to one of the registers and return from exception/switch to user mode follow
- Library wrapper does error processing (sets errno) and regular function return passes execution back to calling program

# System calls – wrapper and system service routine

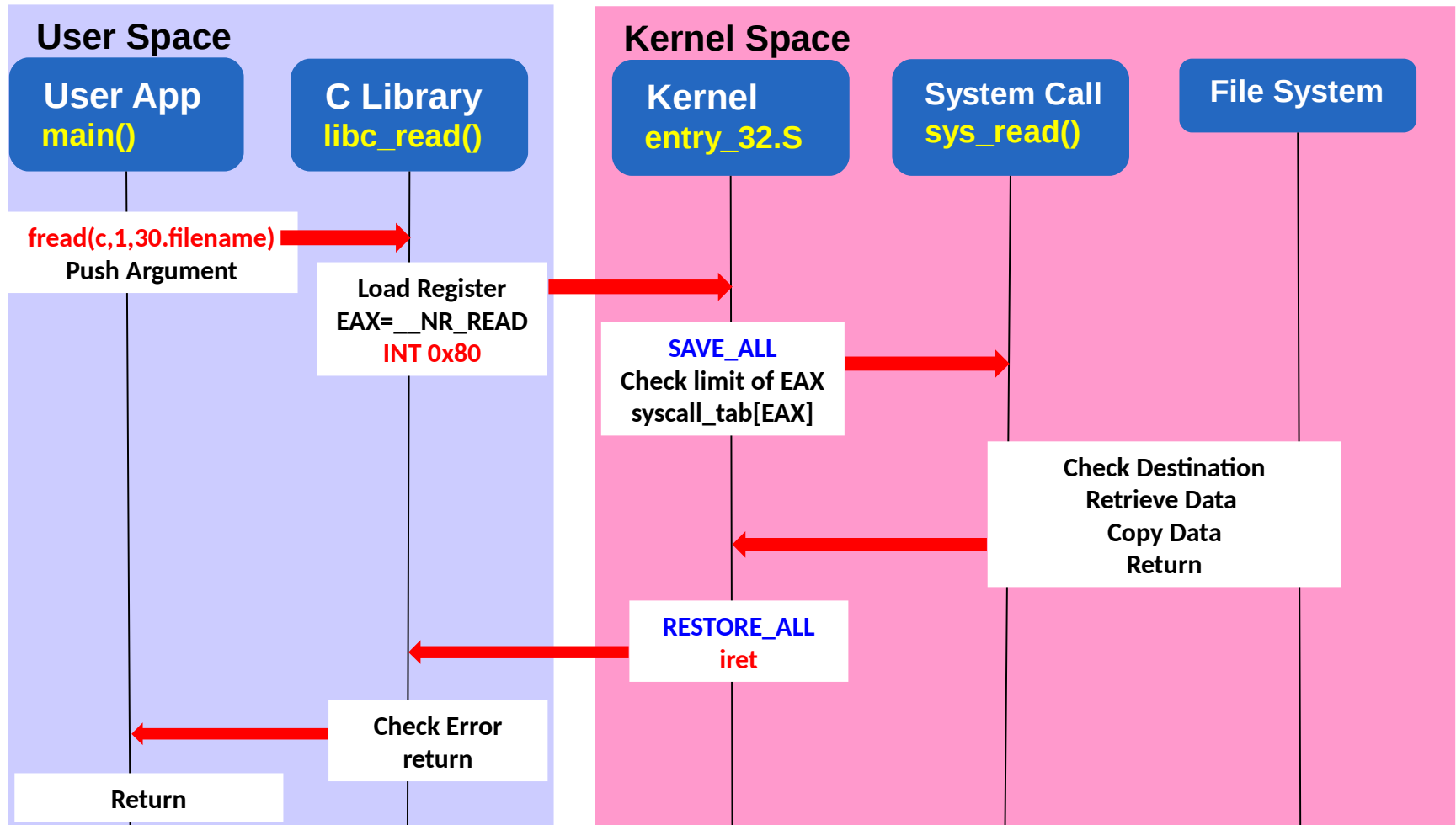


# Parameters placement for selected syscalls (Linux i386)

%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 processing steps



# System call and parameters placement for MIPS architecture

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)	

Actual system call invocation is realized by **SYSCALL** instruction, the numbers are defined in <http://lxr.linux.no/#linux+v3.8.8/arch/mips/include/uapi/asm/unistd.h>

Source: <http://www.linux-mips.org/wiki/Syscall>

# Hello World – the first Linux program ever run on MIPS

```
#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

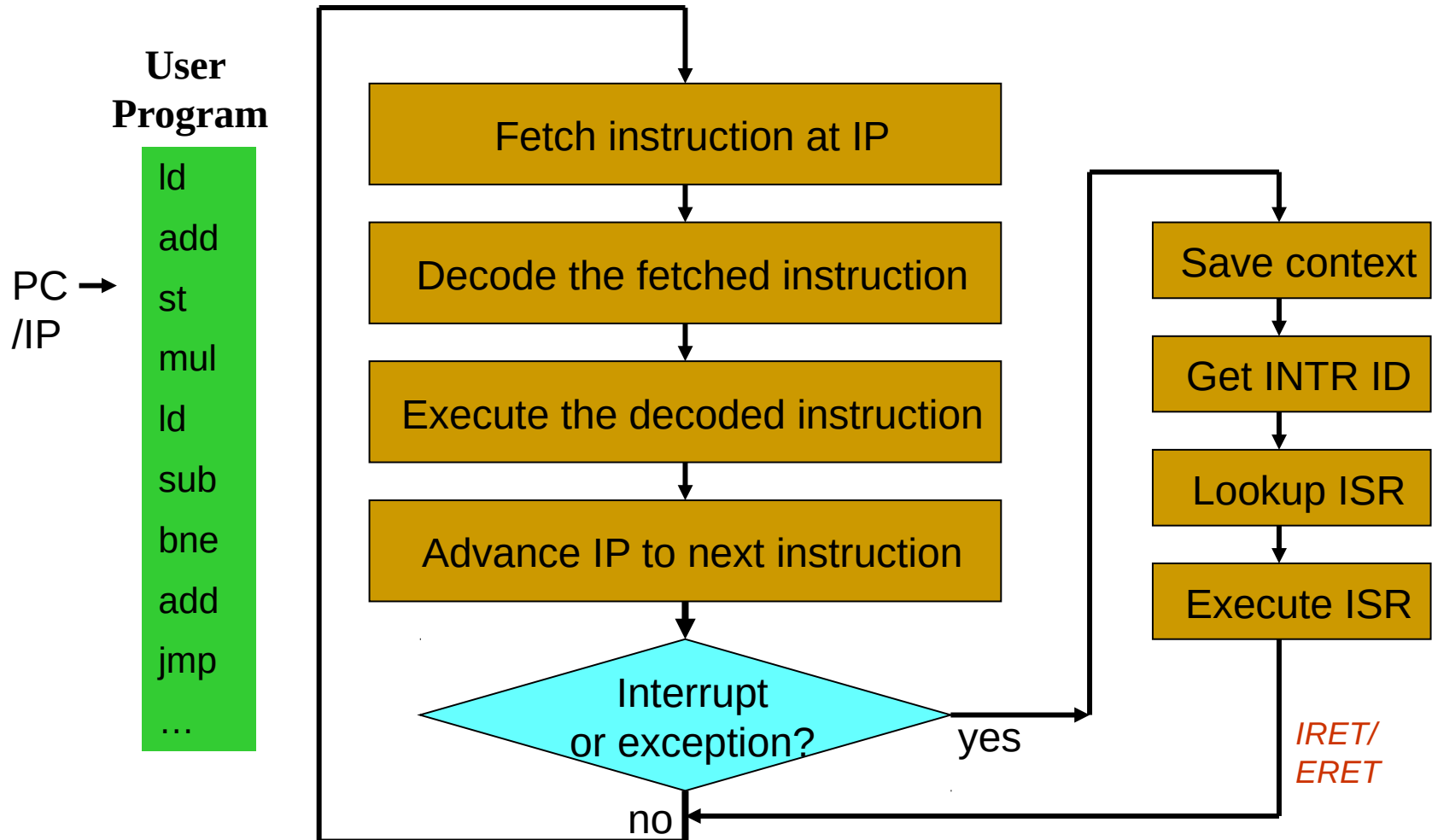
quit:
    li   a0, 0
    li   v0, SYS_exit
syscall

    j    quit
    nop

    END(main)

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

# Interrupts and exceptions in computer execution cycle



## 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;



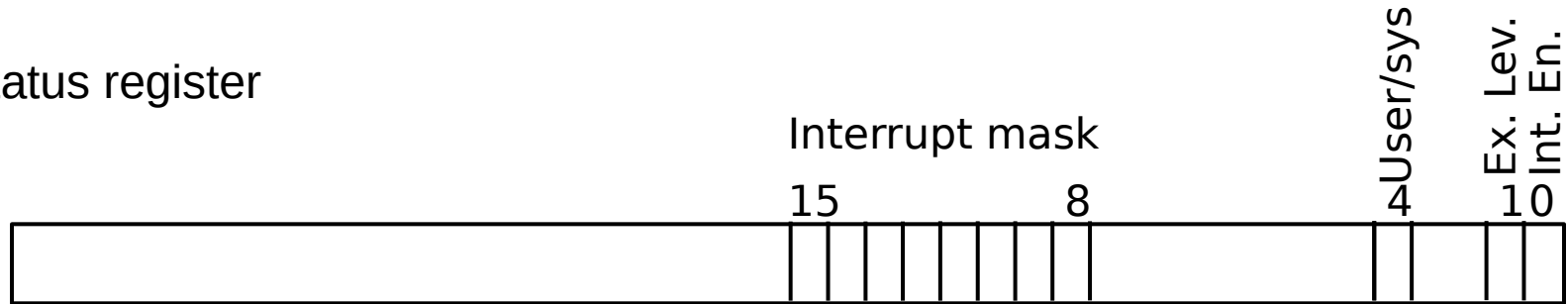
## Exceptions sources on MIPS32 - continued

- 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;

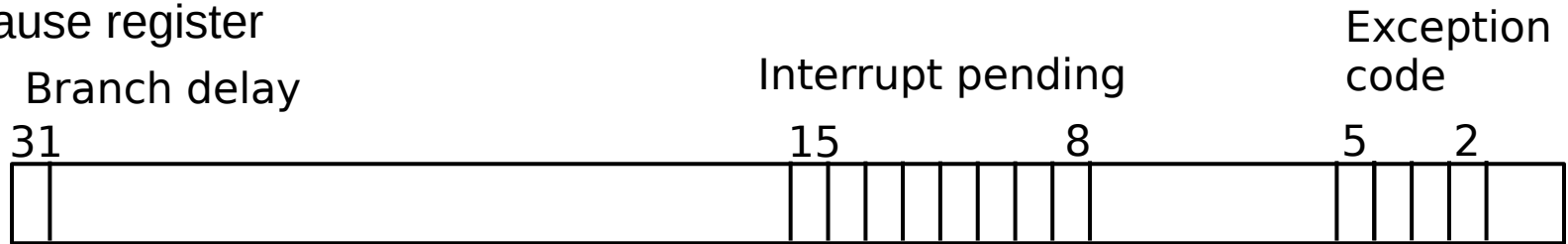
# MIPS – registers for exceptions status and control

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 register)
- 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 from 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.)