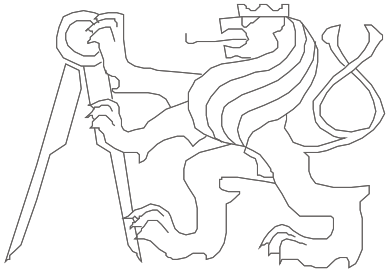
Computer Architectures

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



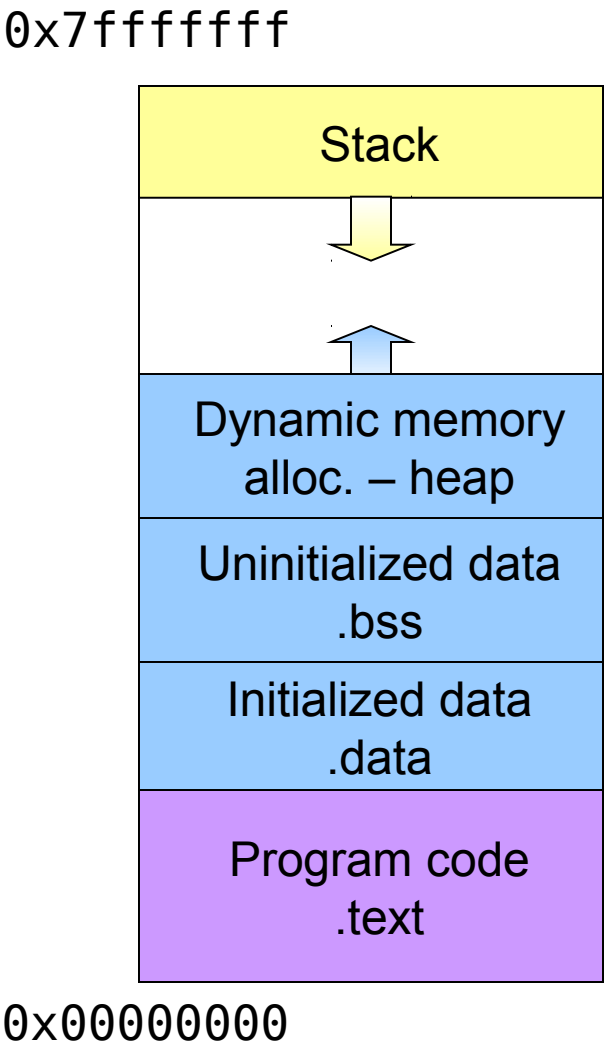
Czech Technical University in Prague, Faculty of Electrical Engineering

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

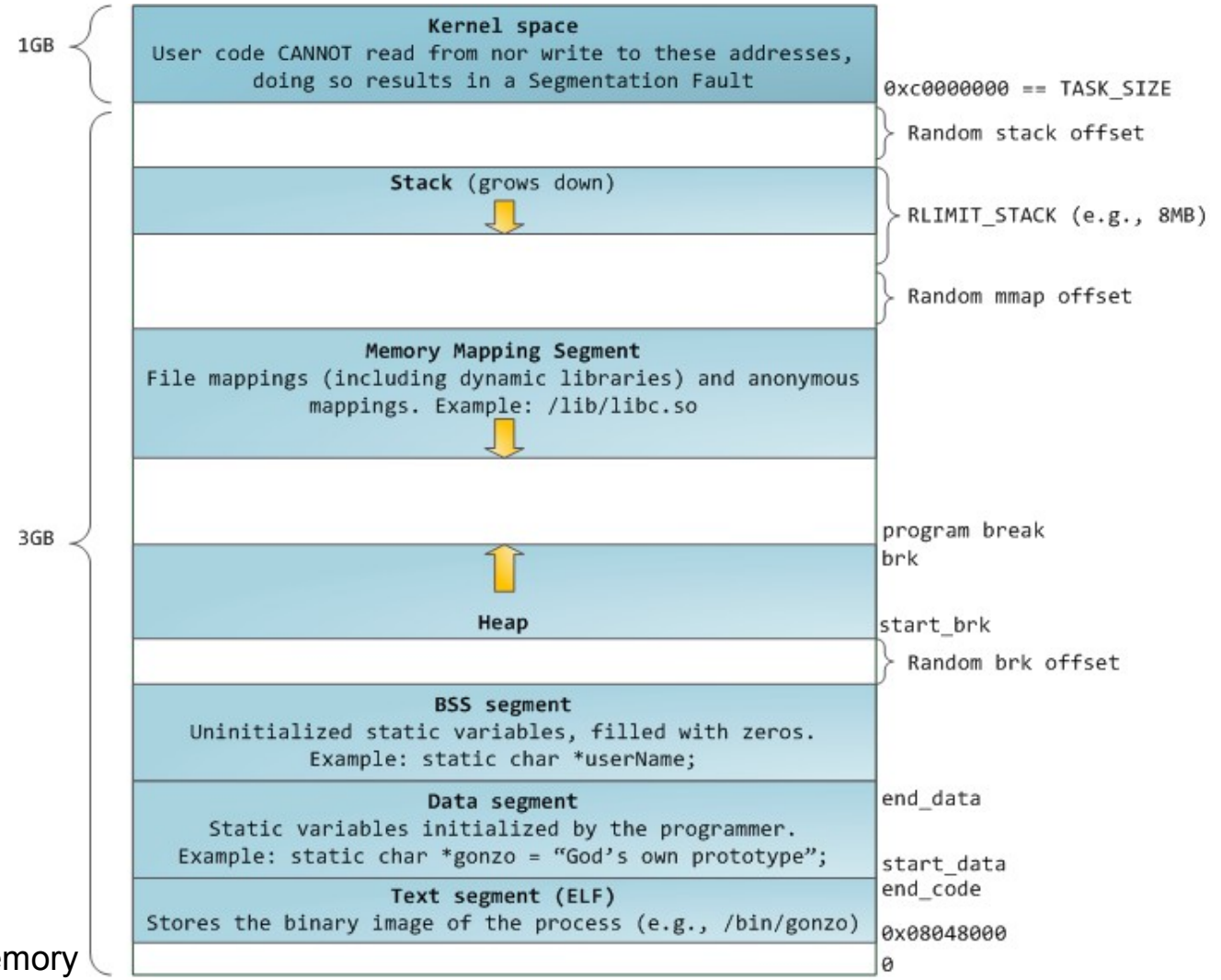
    (x86 Babylonian confusion of tongues - cdecl, syscall, optlink, pascal, register, stdcall, fastcall, safecall, thiscall MS/others)
  - Stack frames allocated space for local variables and C-library alloca()
- System calls (requests a service from OS kernel)
  - Control transfer (switching) from user 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



- 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 neded
- 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

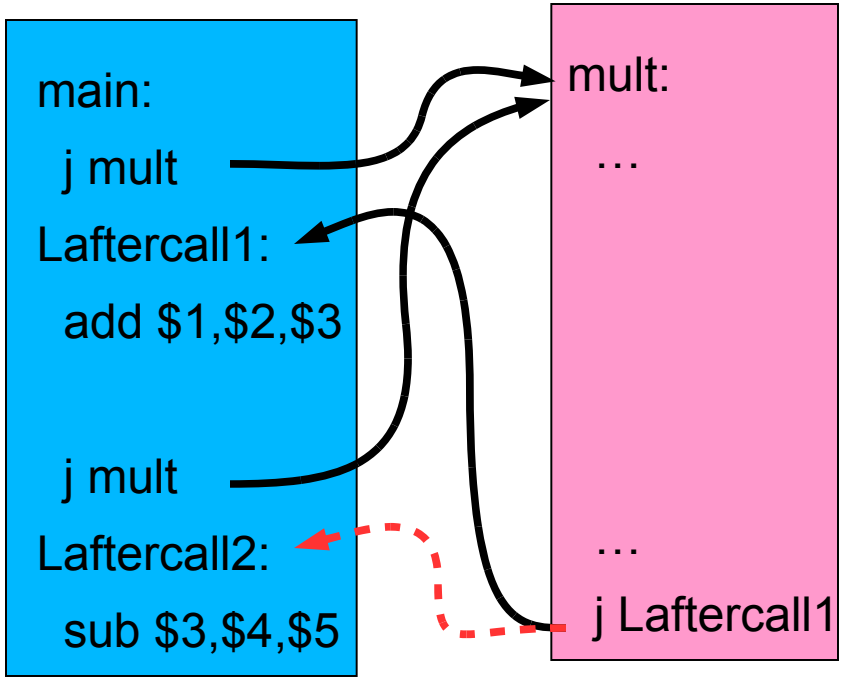
- a0 – a3: arguments (registers $4 – $7)
- v0, v1: registers to hold function result value ($2 and $3)
- t0 – t9: temporary/clobberable registers ($8-$15,$24,$25)
  - callee function can use them as it needs without saving
- at: temporary register for complex assembly constructs ($1)
- k0, k1: reserved 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

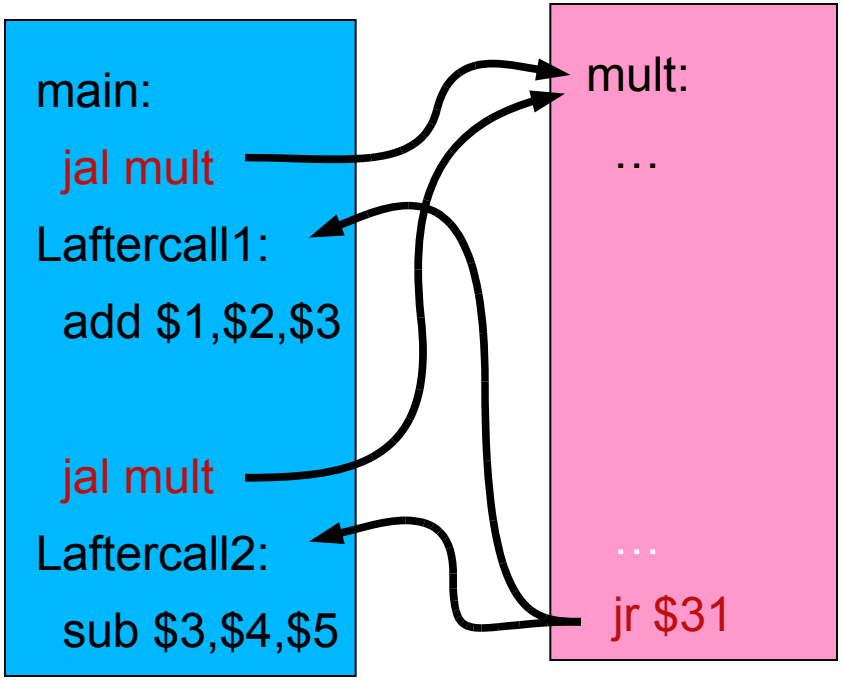
MIPS: call instruction and return

- 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

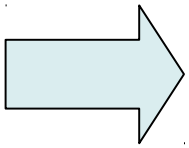
on the contrary, call using register **ra** allows you to call a subroutine from as many locations as needed

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 << 2)
Syntax:	jal target
Encoding:	0000 11ii iiii iiii iiii iiii iiii iiii

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

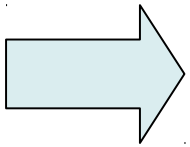
MIPS: Caller and Callee with Parameters Passing

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

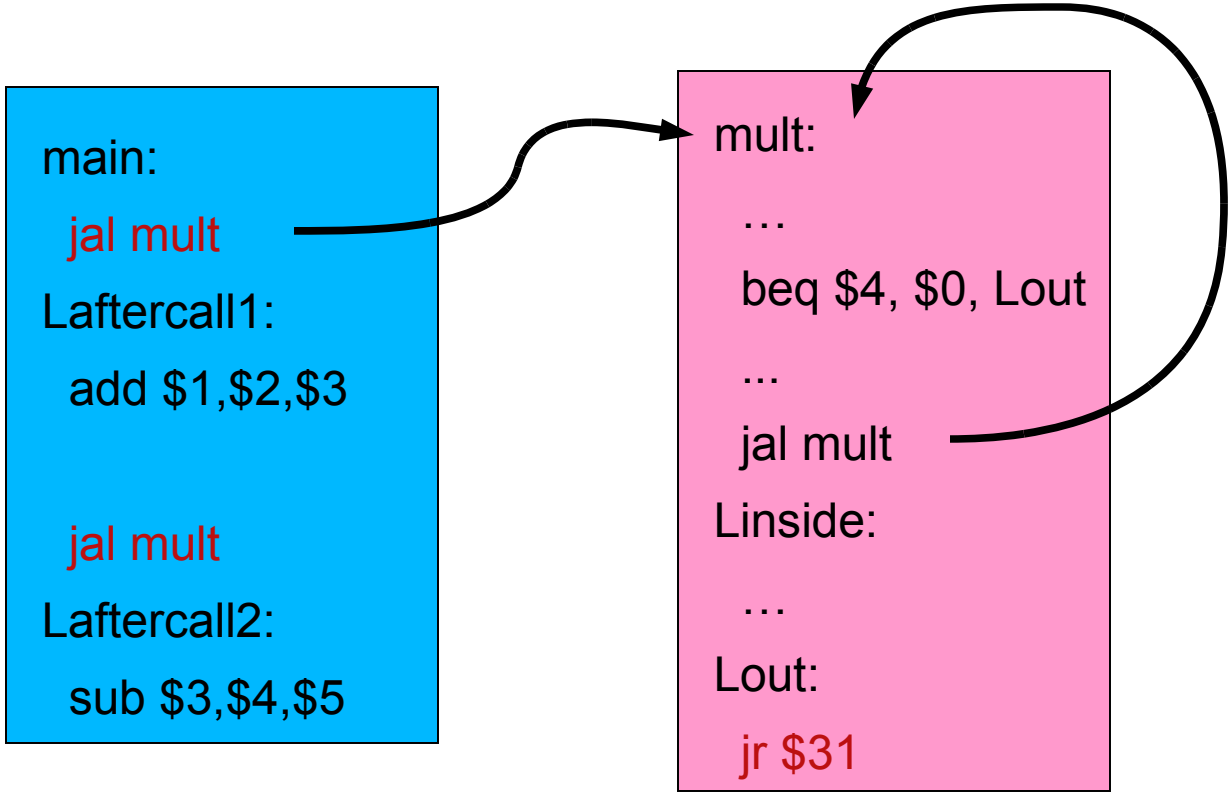
leaf_fun:	
addi $sp, $sp, -4 sw $s0, 0($sp)	Save $s0 on stack

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

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 **s***X*

Recursive function or function calling another one

C language source code:

```
int fact (int n)
  {
    if (n < 1) return f;
    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
```

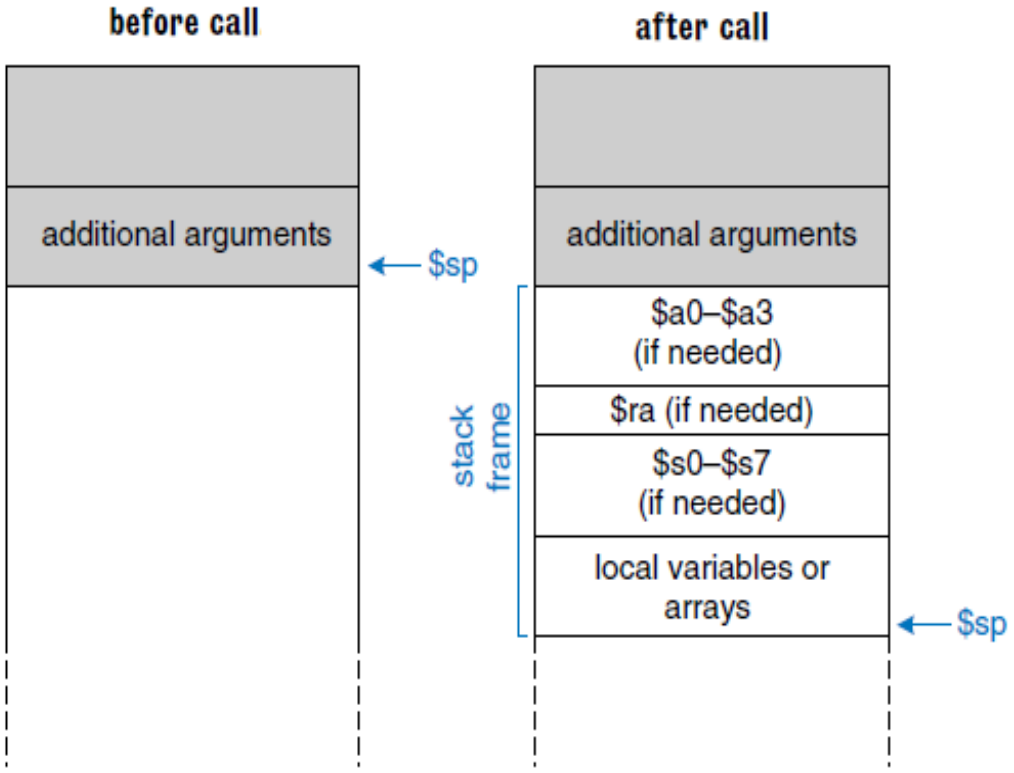
MIPS Calling Convention and ABI

- 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 … $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
```

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

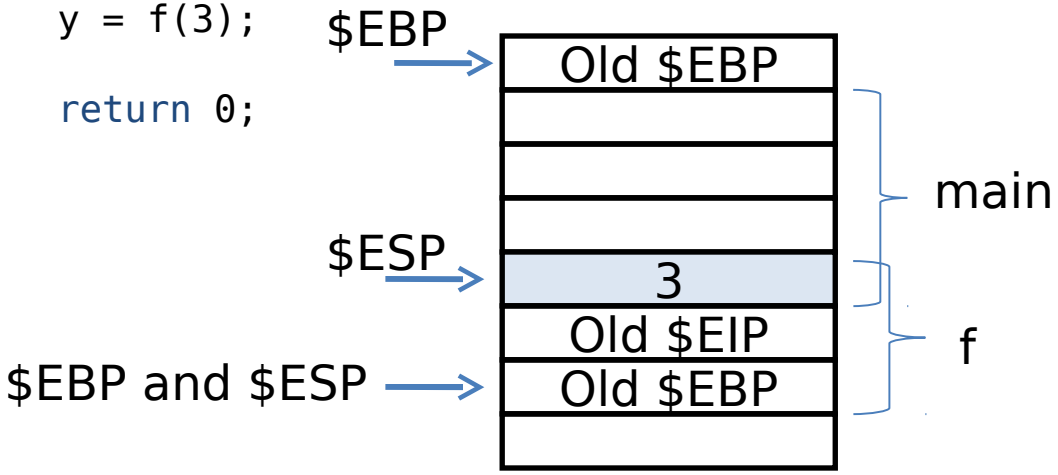
```c
#include <stdio.h>

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
        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

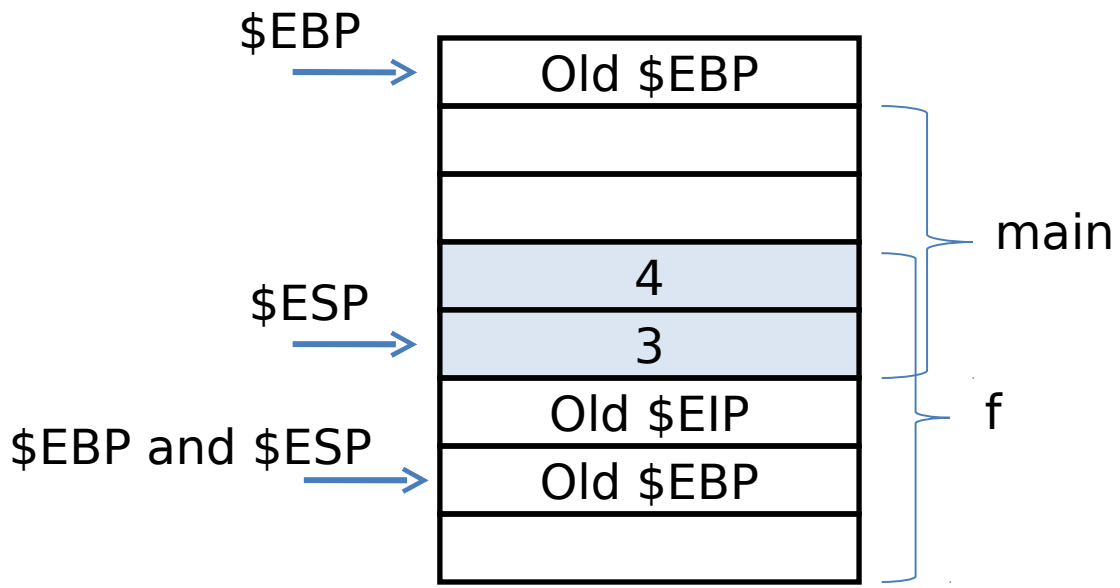
x86: calling function with two parameters

```
#include <stdio.h>

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

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

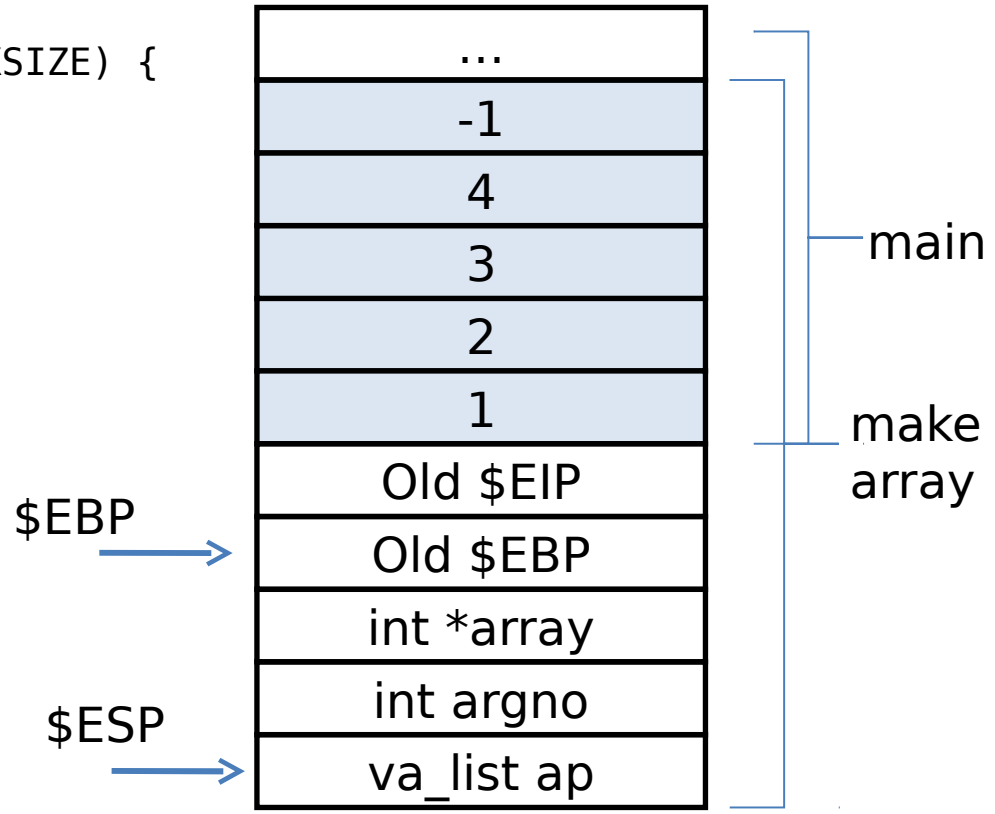
```
f:      pushl   %ebp
        movl    %esp, %ebp
        movl    12(%ebp), %eax
        addl    8(%ebp), %eax
        leave
        ret
main:   pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        subl    $16, %esp
        movl    $4, 4(%esp)
       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

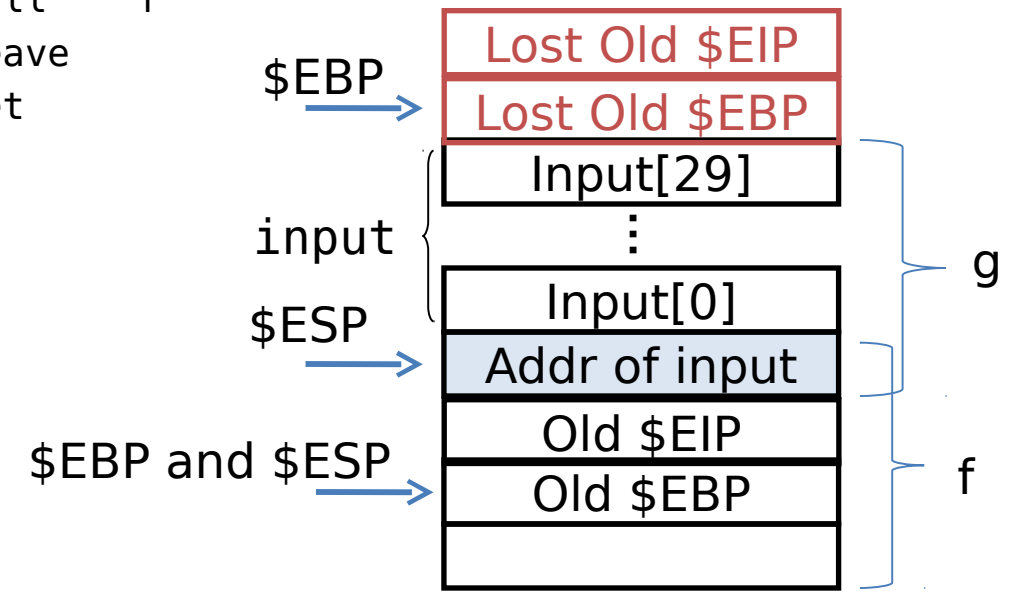
```
int *p;
int i;
p = makearray(1,2,3,4,-1);
for(i=0;i<5;i++)
  printf("%d\n", p[i]);
```

Stack/buffer overflow error example

```
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: 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;
}
```

```
_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
```

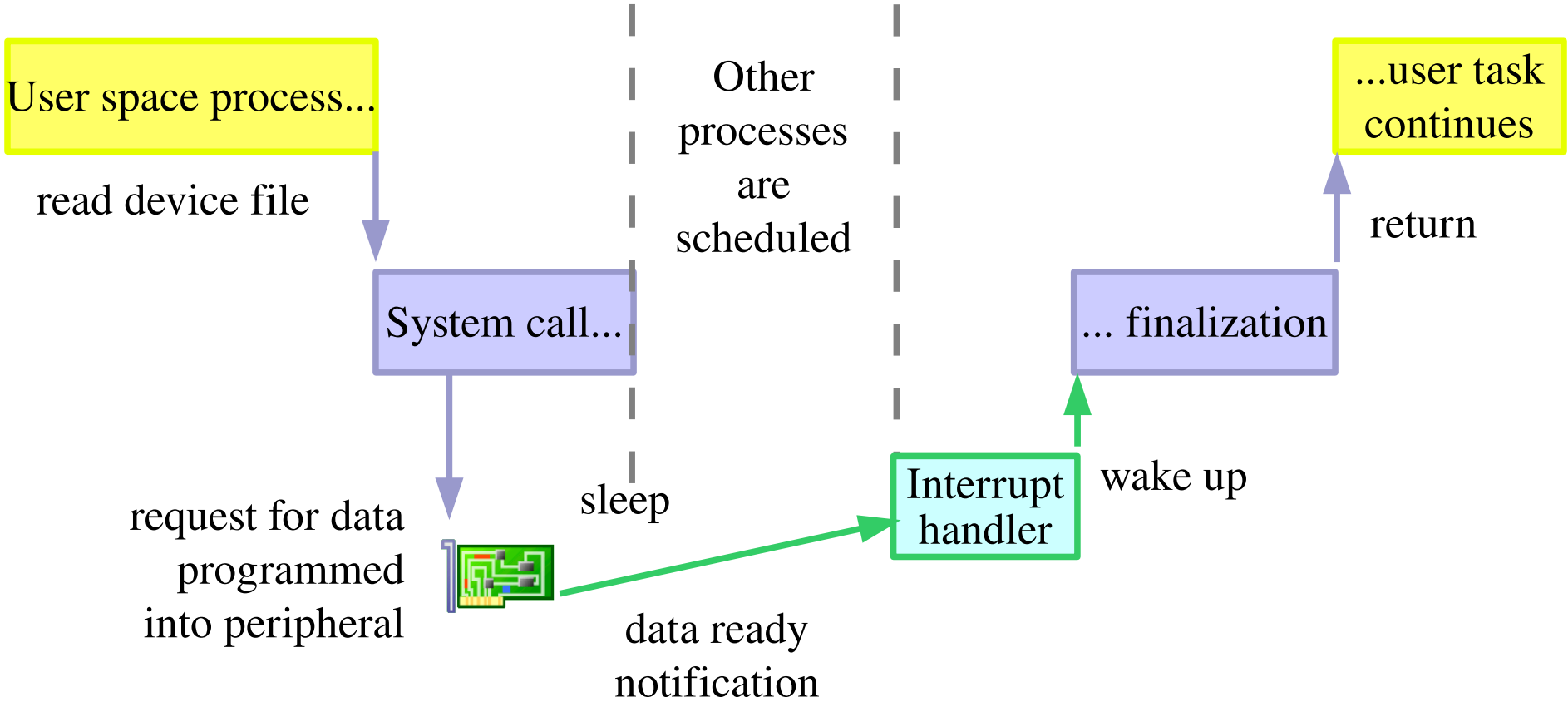
```
_main:
pushl %ebp              ebp saved on stack, push modifies esp
movl  %esp, %ebp        esp to ebp
andl  $-16, %esp        allign stack to 16-bytes
subl  $48, %esp         esp = esp - 48 (allocate space)
call  ___main
movl  $6, 20(%esp)      the last argument
movl  $5, 16(%esp)      the fifth 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
```

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)

System call – repeated from external events lecture

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

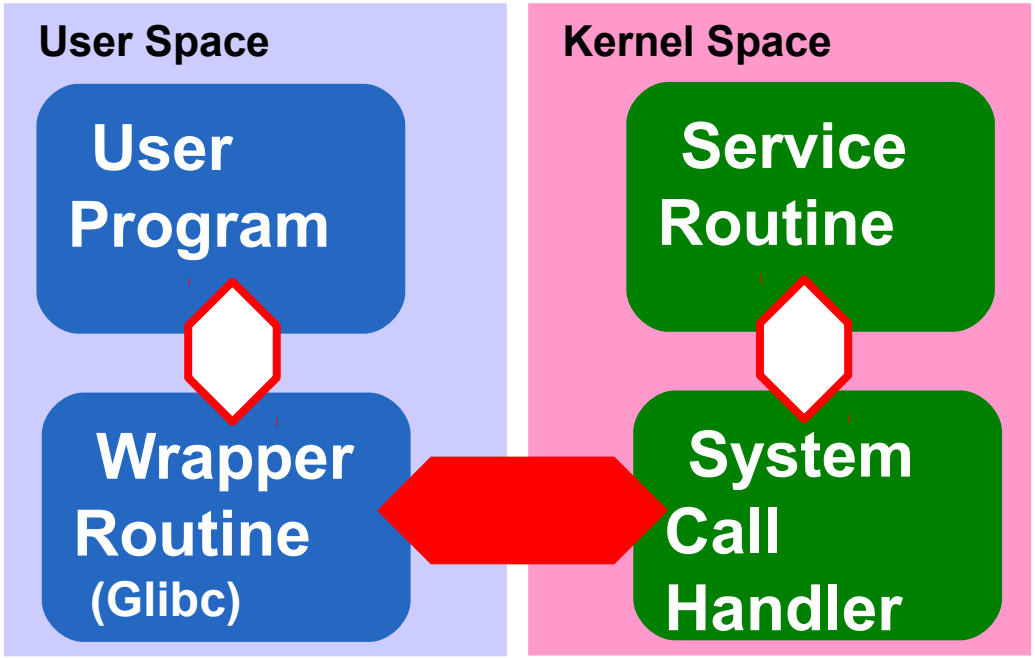


source: Free Electrons: Kernel, drivers and embedded Linux development http://free-electrons.com

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 Number

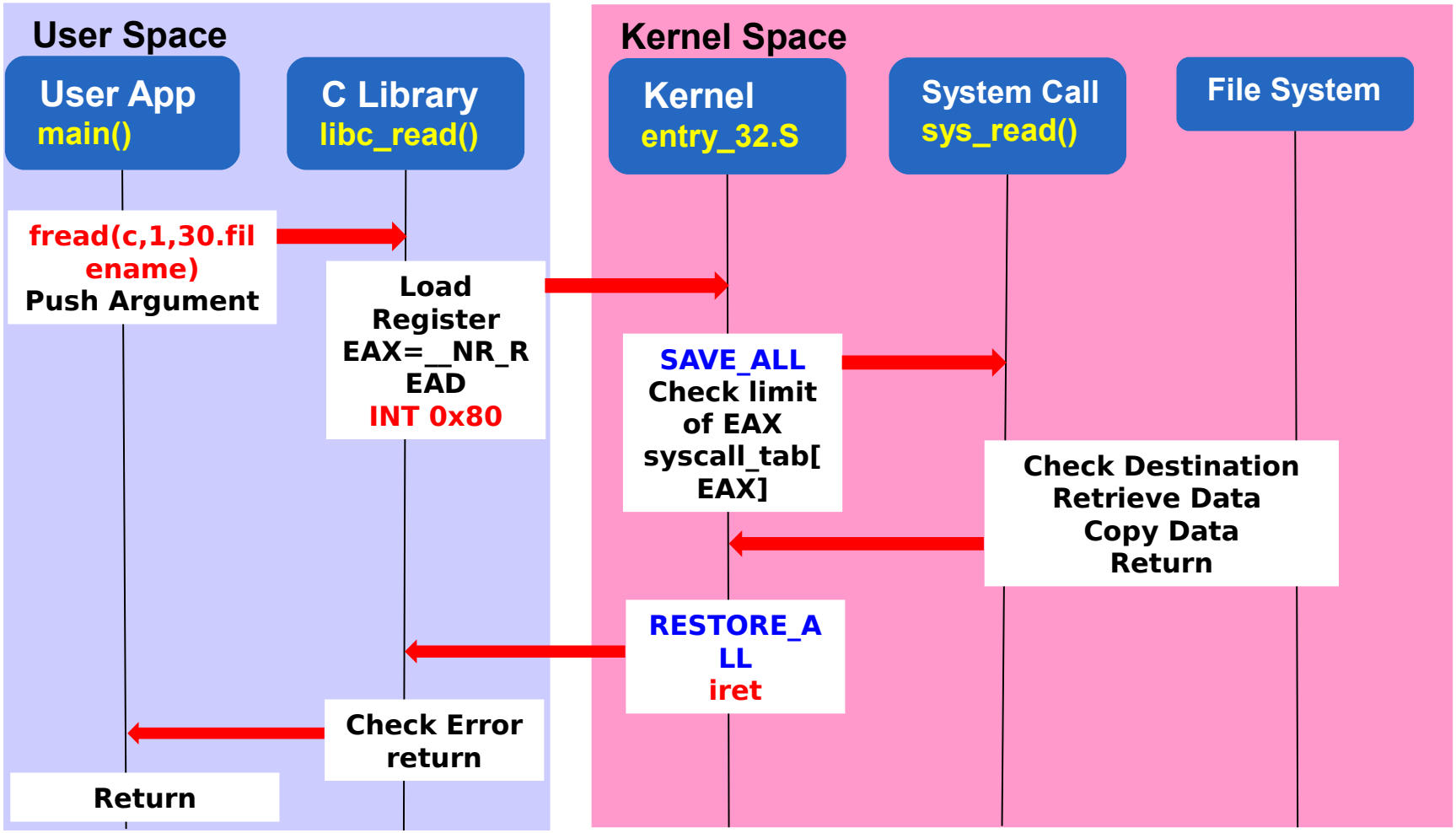
System Call Name

First parameter

Second parameter

Third parameter

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"
```