

System Calls Management

Advanced Operating Systems and Virtualization

Alessandro Pellegrini

A.Y. 2018/2019



SAPIENZA
UNIVERSITÀ DI ROMA

trap_init()

```
gate_desc idt_table[NR_VECTORS] __page_aligned_bss;
```

```
void __init trap_init(void) {  
    set_intr_gate(X86_TRAP_DE, divide_error);  
    set_intr_gate_ist(X86_TRAP_NMI, &nmi, NMI_STACK);  
    set_system_intr_gate(X86_TRAP_OF, &overflow);  
    set_intr_gate(X86_TRAP_BR, bounds);  
    set_intr_gate(X86_TRAP_UD, invalid_op);  
    set_intr_gate(X86_TRAP_NM, device_not_available);  
    set_task_gate(X86_TRAP_DF, GDT_ENTRY_DOUBLEFAULT_TSS);  
    set_intr_gate_ist(X86_TRAP_DF, &double_fault,  
                     DOUBLEFAULT_STACK);  
  
    set_intr_gate(X86_TRAP_OLD_MF,  
                  coprocessor_segment_overrun);  
    set_intr_gate(X86_TRAP_TS, invalid_TSS);  
    ...  
    set_system_trap_gate(SYSCALL_VECTOR, &system_call);  
    ...  
}
```

← 0x80



Userspace Kernel API: System Calls

- For Linux (same for Windows), the gate for on-demand access (via software traps) to the kernel is only one
- For i386 machines the corresponding software traps are:
 - 0x80 for LINUX
 - 0x2E for Windows
- The software module associated with the on-demand access GATE implements a *dispatcher* that is able to trigger the activation of the specific system call targeted by the application

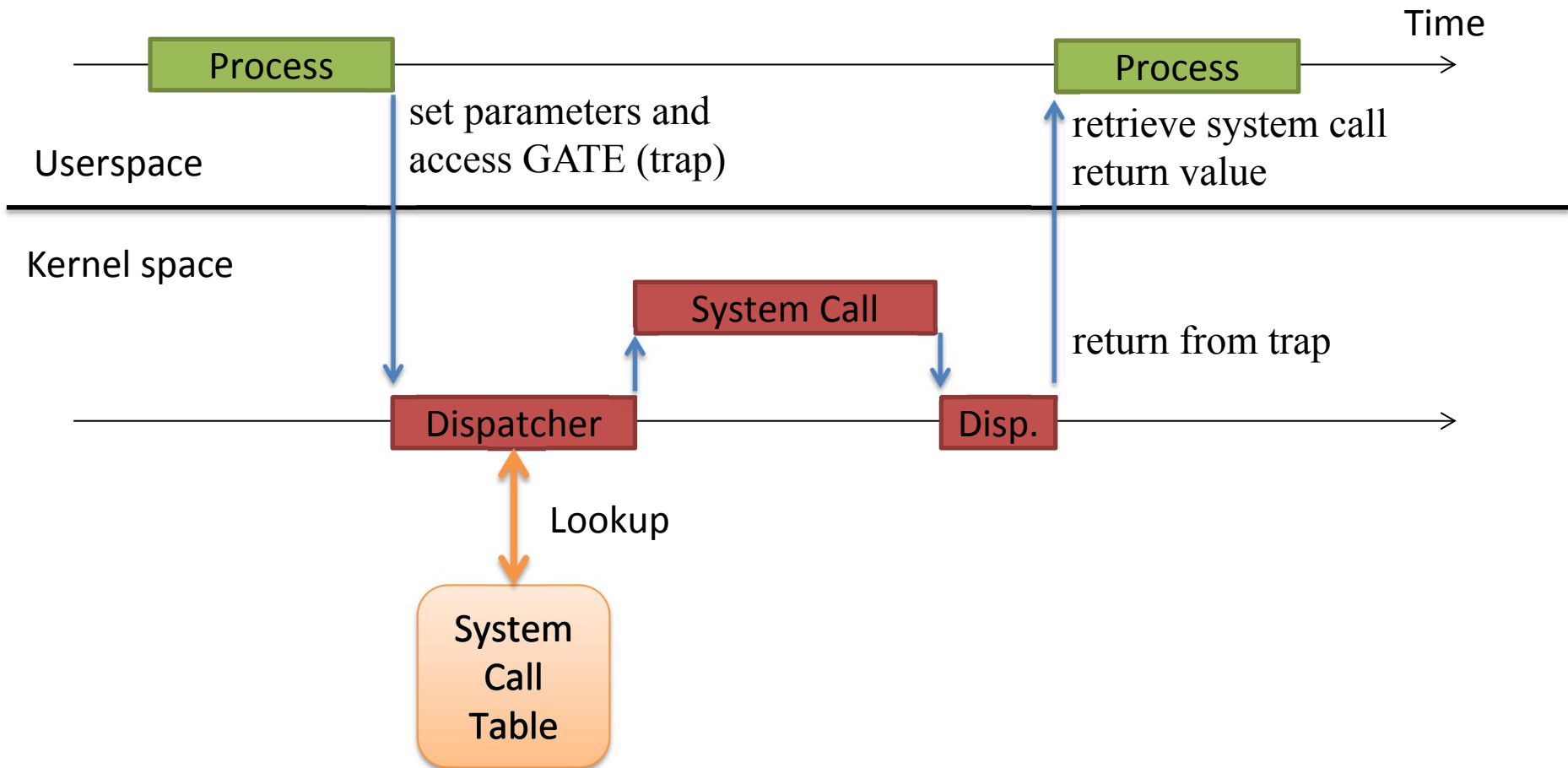


System Call Dispatching

- The main data structure is the *system call* table
- Each entry of the table points to a kernel-level function, activated by the dispatcher
- To access the correct entry, the dispatcher needs as input the system call number (provided in a CPU register)
- The code is used to identify the target entry within the system call table
- The system call is activated via an indirect call
- The return value is returned in a register



Dispatcher Mechanism



Compile-Time Syscall Interface (2.4)

- This is all based on macros
 - Macros for standard formats are in `include/asm-xx/unistd.h` (or `asm/unistd.h`)
- There we find:
 - System call numerical codes
 - They are numbers used to invoke a syscall for userspace
 - They are a displacement in the syscall table for kernel space
 - Standard macros to let userspace access the gate to the Kernel
 - There is a macro for each range of parameters, from 0 to 6



Syscall codes (2.4.20)

```
/*
 * This file contains the system call
 * numbers.
 */

#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
#define __NR_write 4
#define __NR_open 5
#define __NR_close 6
#define __NR_waitpid 7
#define __NR_creat 8
#define __NR_link 9
#define __NR_unlink 10
#define __NR_execve 11
#define __NR_chdir 12
.....
#define __NR_fallocate 324
```



Macro for a 0-Parameters Syscall

```
#define __syscall0(type,name) \  
type name(void) \  
{ \  
    long __res; \  
    __asm__ volatile ("int $0x80" \  
        : "=a" (__res) \  
        : "0" (__NR_##name)); \  
    __syscall_return(type, __res); \  
}
```

Example syscall: `fork()`



Return from a syscall

```
/* user-visible error numbers are in the range -1 - -124:
   see <asm-i386/errno.h> */

#define __syscall_return(type, res) \
do { \
    if ((unsigned long)(res) >= (unsigned long)(-125)) { \
        errno = -(res); \
        res = -1; \
    } \
    return (type) (res); \
} while (0)
```

← Only if res in [-1, -124]

← What's that?!



Macro for a 1-Parameter Syscall

```
#define __syscall1(type,name,type1,arg1) \
type name(type1 arg1) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
: "=a" (__res) \
: "0" (__NR_##name), "b" ((long) (arg1))) ; \
__syscall_return(type, __res); \
}
```

Example syscall: `close()`



Macro for a 6-Parameters Syscall

```
#define __syscall6(type,name,type1,arg1,type2,arg2,\
                type3,arg3,type4,arg4,type5,arg5,type6,arg6) \
type name (type1 arg1,type2 arg2,type3 arg3,\
        type4 arg4,type5 arg5,type6 arg6) \
{ \
    long __res; \
    __asm__ volatile ( \
        "push %%ebp ; movl %%eax,%%ebp ;" \
        "movl %1,%%eax ; int $0x80 ; pop %%ebp" \
        : "=a" (__res) \
        : "i" (__NR_##name), "b" ((long)(arg1)), \
          "c" ((long)(arg2)), "d" ((long)(arg3)), \
          "S" ((long)(arg4)), "D" ((long)(arg5)), \
          "0" ((long)(arg6)) \
        ); \
    __syscall_return(type,__res); \
}
```

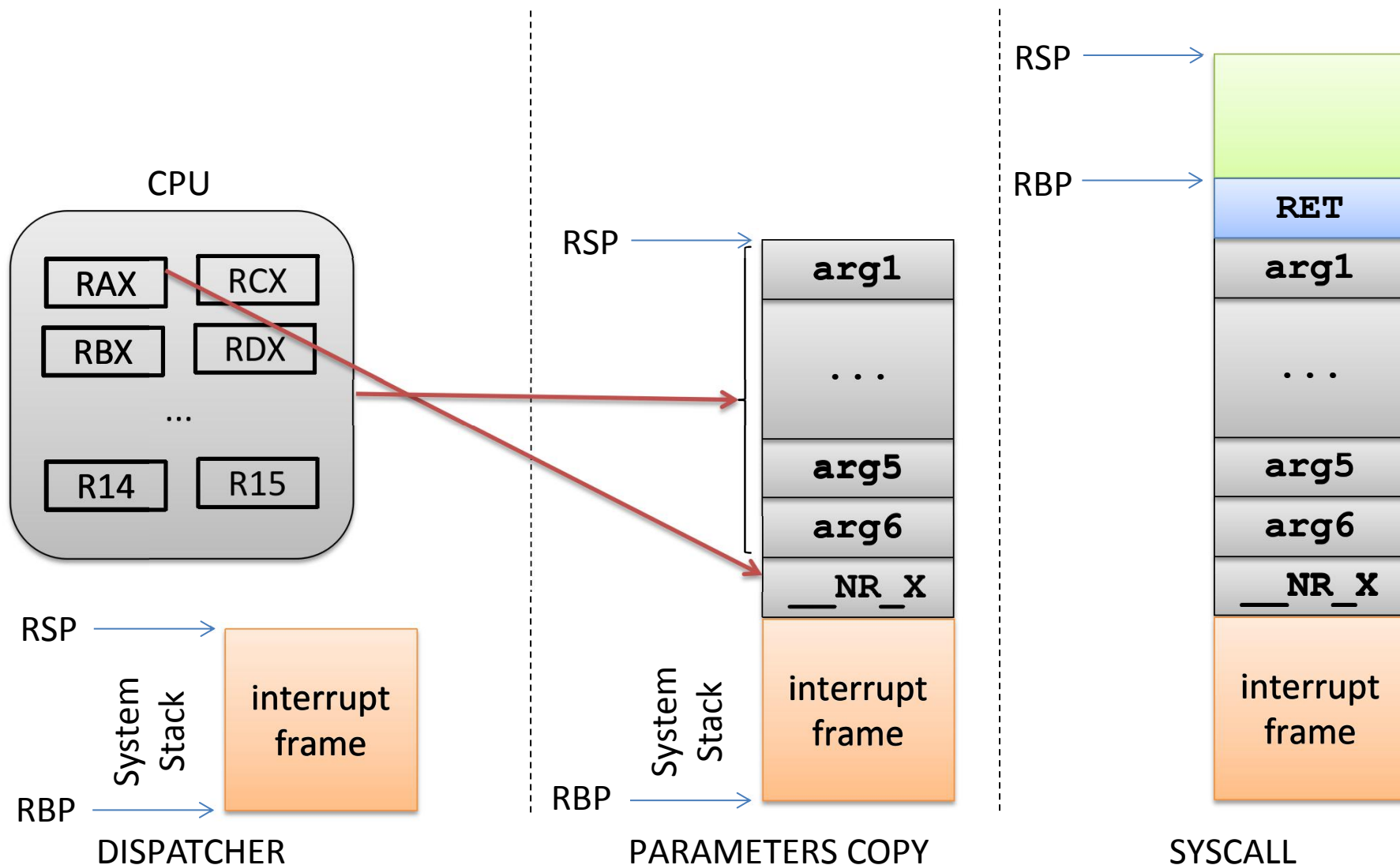


Dispatcher Activities

- Once gained control, the dispatcher takes a complete snapshot of CPU registers
- The snapshot is taken within the system-level stack
- Then the system call is invoked as a subroutine call (via a `call`)
- The system call retrieves the parameters from stack via the base pointer (remember `asm linkage?`)



Dispatcher Activities



CPU Stack (i386)

pt_regs	/*				
	*	0 (%esp)	-	%ebx	arguments
	*	4 (%esp)	-	%ecx	
	*	8 (%esp)	-	%edx	
	*	C (%esp)	-	%esi	
	*	10 (%esp)	-	%edi	
	*	14 (%esp)	-	%ebp	Interrupt frame
	*	18 (%esp)	-	%eax	
	*	1C (%esp)	-	%ds	
	*	20 (%esp)	-	%es	
	*	24 (%esp)	-	orig_eax	
	*	28 (%esp)	-	%eip	
	*	2C (%esp)	-	%cs	
	*	30 (%esp)	-	%eflags	
	*	34 (%esp)	-	%oldesp	
	*	38 (%esp)	-	%oldss	
	*/				

← Syscall number



Syscall Dispatcher (i386)

```
ENTRY(system_call)
    pushl %eax    # save eax
    SAVE_ALL
    GET_CURRENT(%ebx)
    testb $0x02, tsk_ptrace(%ebx) # PT_TRACESYS
    jne tracesys
    cmpl $(NR_syscalls), %eax
    jae badsys
    call *SYMBOL_NAME(sys_call_table)(, %eax, 4)
    movl %eax, EAX(%esp)    # save the return value
ENTRY(ret_from_sys_call)
    cli            # need_resched and signals atomic test
    cmpl $0, need_resched(%ebx)
    jne reschedule
    cmpl $0, sigpending(%ebx)
    jne signal_return
restore_all:
    RESTORE_ALL
```



`syscall()`

- This is a construct introduced in Kernel 2.6 for the Pentium 3 chip
- Implemented through glibc (stdlib.h)
- It triggers a trap to execute a generic system call
- The first argument is the system call number
- The other parameters are the input for the system call code
- Based on new x86 instructions: `sysenter/sysexit` or `syscall/sysret` (initially for AMD chips)



Fast syscall Path

SYSENTER/SYSCALL

based on model-specific registers

CS register set to the value of (SYSENTER_CS_MSR)

EIP register set to the value of (SYSENTER_EIP_MSR)

SS register set to the sum of (8 plus the value in SYSENTER_CS_MSR)

ESP register set to the value of (SYSENTER_ESP_MSR)

SYSEXIT/SYSRET

based on model-specific registers

CS register set to the sum of (16 plus the value in SYSENTER_CS_MSR)

EIP register set to the value contained in the EDX register

SS register set to the sum of (24 plus the value in SYSENTER_CS_MSR)

ESP register set to the value contained in the ECX register



Model-Specific Registers for Syscalls

```
/include/asm/msr.h:
```

```
#define MSR_IA32_SYSENTER_CS      0x174  
#define MSR_IA32_SYSENTER_ESP    0x175  
#define MSR_IA32_SYSENTER_EIP    0x176
```

```
/arch/x86/kernel/cpu/common.c:
```

```
wrmsr(MSR_IA32_SYSENTER_CS, __KERNEL_CS, 0);  
wrmsr(MSR_IA32_SYSENTER_ESP, tss->esp0, 0);  
wrmsr(MSR_IA32_SYSENTER_EIP, (unsigned long)  
      sysenter_entry, 0);
```

Again based on `rdmsr` and `wrmsr`



x64 Calling Conventions (syscalls)

```
/*  
 * Register setup:  
 * rax  system call number  
 * rdi  arg0  
 * rcx  ret.address for syscall/sysret, userspace arg3  
 * rsi  arg1  
 * rdx  arg2  
 * r10  arg3 (--> to rcx for userspace)  
 * r8   arg4  
 * r9   arg5  
 * r11  eflags for syscall/sysret, temporary for C  
 * r12-r15,rbp,rbx saved by C code, not touched.  
 *  
 * Interrupts are off on entry.  
 * Only called from user space.  
 */
```



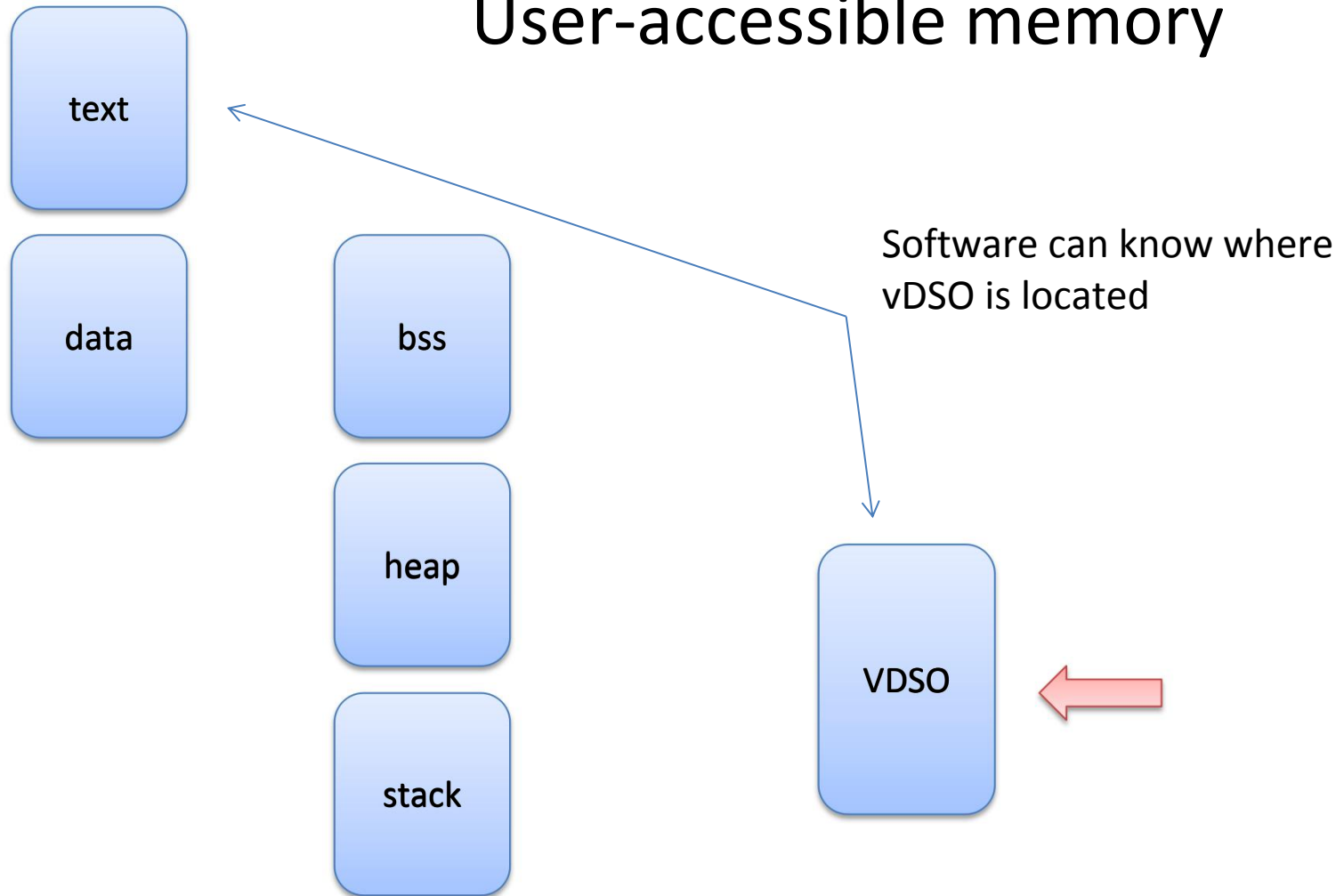
Virtual Dynamic Shared Object (vDSO)

- Syscall entry/exit points are set by the Kernel
- Few memory pages are created and made visible to all processes' address spaces when they are initialized
- These processes find the actual code for the syscall entry/exit mechanism
- For i386 the definition is (up to Kernel 2.6.23) in `arch/i386/kernel/vsyscall-sysenter.S`
- In later versions, it's become an actual shared library. The source tree is at `/source/arch/x86/vdso` and the entry point is thus moved to `/arch/x86/vdso/vdso32/sysenter.S`



Mapping vDSO

User-accessible memory



Exposing vDSO

```
#include <sys/auxv.h>
void *vdso = (uintptr_t) getauxval(AT_SYSINFO_EHDR);
```

The "vDSO" (virtual dynamic shared object) is a small shared library that the kernel automatically maps into the address space of all user-space applications. Applications usually do not need to concern themselves with these details as the vDSO is most commonly called by the C library. This way you can code in the normal way using standard functions and the C library will take care of using any functionality that is available via the vDSO.



vDSO Entry Point

```
__kernel_vsyscall:  
    push %ecx  
    push %edx  
    push %ebp  
    movl %esp,%ebp  
    sysenter  
    nop  
    /* 14: System call restart point is here! */  
    int $0x80  
    /* 16: System call normal return point is here! */  
    pop %ebp  
    pop %edx  
    pop %ecx  
    ret
```



Considerations on the vDSO

- The vDSO Kernel entry point exploits flat addressing to bypass segmentation and the related operations
- It therefore reduces the number of accesses to memory in order to support the change to kernel mode
- Studies show that the reduction of clock cycles for system calls can be in the order of 75%
- It allows randomization: security is enhanced



The syscall Table

- The kernel level system call table is defined in specific files:
 - For Kernel 2.4.20 on i386 it is defined in `arch/i386/kernel/entry.S`
 - For Kernel 2.6 is in `arch/x86/kernel/syscall_table32.S`
 - More recent versions:
`/arch/x86/entry/syscalls/syscall_32.tbl`
- Entries keep a reference to the kernel-level system call implementation
- Typically, the kernel-level name resembles the one used at application level (traditionally `sys_...`)



C Syscall Entry Points (4.20)

- `/arch/x86/entry/syscalls/syscall_64.tbl:`
`0 common read __x64_sys_read`
- `/include/linux/syscalls.h:`
`asmlinkage long sys_read(unsigned int fd, char __user
*buf, size_t count);`
- `/fs/read_write.c:`
`SYSCALL_DEFINE3(read, unsigned int, fd, char __user *,
buf, size_t, count)
{
 return ksys_read(fd, buf, count);
}`



C Syscall Entry Points (4.20)

- `include/linux/syscalls.h`:

```
#define SYSCALL_DEFINE3(name, ...) \
SYSCALL_DEFINEx(3, _##name, __VA_ARGS__)
```



```
#define SYSCALL_DEFINEx(x, sname, ...) \
    SYSCALL_METADATA(sname, x, __VA_ARGS__) \
    __SYSCALL_DEFINEx(x, sname, __VA_ARGS__)
```
- The final macro `__SYSCALL_DEFINEx()` will generate the actual entry point of the system call, with additional protection mechanisms



C Syscall Entry Points (4.20)

```
asmlinkage long sys_read(unsigned int fd, char __user * buf, size_t count)
__attribute__((alias(__stringify(Sys_read))));
```

for security reasons (sign extension)

```
asmlinkage long Sys_read(long int fd, long int buf, long int count)
{
    long ret = SYSC_read((unsigned int) fd, (char __user *) buf, (size_t) count);
    asmlinkage_protect(3, ret, fd, buf, count);
    return ret;
}
```

```
static inline long SYSC_read(unsigned int fd, char __user * buf, size_t count)
{
    return ksys_read(fd, buf, count);
}
```



32/64 bit Paths

