Linux System Call Table

The following table lists the **system calls** for the **Linux** 2.2 kernel. It could also be thought of as an API for the interface between user space and kernel space. My motivation for making this table was to make programming in assembly language easier when using only **system calls** and not the C library (for more information on this topic, go to http://www.linuxassembly.org). On the left are the numbers of the **system calls**. This number will be put in register %eax. On the right of the table are the types of values to be put into the remaining registers before calling the software interrupt 'int 0x80'. After each syscall, an integer is returned in %eax.

For convenience, links go from the "Name" column to the man page for most of the **system calls**. Links to the kernel source file where each **system** call is located are linked to in the column labelled "Source". (You can also download a version of this page which has links directly to the source that is installed on your **system**.) Links to definitions are provided for the parameters that are typedefs or structs.

%eax	Name	Source	%ebx	%ecx	%edx	%esx	%edi
1	sys_exit	kernel/exit.c	int	-	-	-	-
2	sys_fork	arch/i386/kernel/process.c	struct pt_regs	-	-	-	-
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	int	-	-
6	sys_close	fs/open.c	unsigned int	-	-	-	-
7	sys_waitpid	kernel/exit.c	pid_t	unsigned int *	int	-	-
8	sys_creat	fs/open.c	const char *	int	-	-	-
9	sys_link	fs/namei.c	const char *	const char *	-	-	-
10	sys_unlink	fs/namei.c	const char *	-	-	-	-
11	sys_execve	arch/i386/kernel/process.c	struct pt_regs	-	-	-	-
12	sys_chdir	fs/open.c	const char *	-	-	-	-
13	sys_time	kernel/time.c	int *	-	-	-	-
14	sys_mknod	fs/namei.c	const char *	int	dev_t	-	-
15	sys_chmod	fs/open.c	const char *	mode_t	-	-	-
16	sys_lchown	fs/open.c	const char *	uid_t	gid_t	-	-
18	sys_stat	fs/stat.c	char *	struct __old_kernel_stat *	-	-	-
19	sys_lseek	fs/read_write.c	unsigned int	off_t	unsigned int	-	-
20	sys_getpid	kernel/sched.c	-	-	-	-	-
21	sys_mount	fs/super.c	char *	char *	char *	-	-
22	sys_oldumount	fs/super.c	char *	-	-	-	-
23	sys_setuid	kernel/sys.c	uid_t	-	-	-	-
24	sys_getuid	kernel/sched.c	-	-	-	-	-
25	sys_stime	kernel/time.c	int *	-	-	-	-
26	sys_ptrace	arch/i386/kernel/ptrace.c	long	long	long	long	-
27	sys_alarm	kernel/sched.c	unsigned int	-	-	-	-
28	sys_fstat	fs/stat.c	unsigned int	struct __old_kernel_stat *	-	-	-
29	sys_pause	arch/i386/kernel/sys_i386.c	-	-	-	-	-
30	sys_utime	fs/open.c	char *	struct utimbuf *	-	-	-
33	sys_access	fs/open.c	const char *	int	-	-	-
34	sys_nice	kernel/sched.c	int	-	-	-	-
36	sys_sync	fs/buffer.c	-	-	-	-	-
37	sys_kill	kernel/signal.c	int	int	-	-	-
38	sys_rename	fs/namei.c	const char *	const char *	-	-	-
39	sys_mkdir	fs/namei.c	const char *	int	-	-	-
40	sys_rmdir	fs/namei.c	const char *	-	-	-	-
41	sys_dup	fs/fcntl.c	unsigned int	-	-	-	-
42	sys_pipe	arch/i386/kernel/sys_i386.c	unsigned long *	-	-	-	-
43	sys_times	kernel/sys.c	struct tms *	-	-	-	-
45	sys_brk	mm/mmap.c	unsigned long	-	-	-	-

46	sys_setgid	kernel/sys.c	gid_t	-	-	-	-
47	sys_getgid	kernel/sched.c	-	-	-	-	-
48	sys_signal	kernel/signal.c	int	__sighandler_t	-	-	-
49	sys_geteuid	kernel/sched.c	-	-	-	-	-
50	sys_getegid	kernel/sched.c	-	-	-	-	-
51	sys_acct	kernel/acct.c	const char *	-	-	-	-
52	sys_umount	fs/super.c	char *	int	-	-	-
54	sys_ioctl	fs/ioctl.c	unsigned int	unsigned int	unsigned long	-	-
55	sys_fcntl	fs/fcntl.c	unsigned int	unsigned int	unsigned long	-	-
57	sys_setpgid	kernel/sys.c	pid_t	pid_t	-	-	-
59	sys_olduname	arch/i386/kernel/sys_i386.c	struct oldold_utsname *	-	-	-	-
60	sys_umask	kernel/sys.c	int	-	-	-	-
61	sys_chroot	fs/open.c	const char *	-	-	-	-
62	sys_ustat	fs/super.c	dev_t	struct ustat *	-	-	-
63	sys_dup2	fs/fcntl.c	unsigned int	unsigned int	-	-	-
64	sys_getppid	kernel/sched.c	-	-	-	-	-
65	sys_getpgrp	kernel/sys.c	-	-	-	-	-
66	sys_setsid	kernel/sys.c	-	-	-	-	-
67	sys_sigaction	arch/i386/kernel/signal.c	int	const struct old_sigaction *	struct old_sigaction *	-	-
68	sys_sgetmask	kernel/signal.c	-	-	-	-	-
69	sys_ssetmask	kernel/signal.c	int	-	-	-	-
70	sys_setreuid	kernel/sys.c	uid_t	uid_t	-	-	-
71	sys_setregid	kernel/sys.c	gid_t	gid_t	-	-	-
72	sys_sigsuspend	arch/i386/kernel/signal.c	int	int	old_sigset_t	-	-
73	sys_sigpending	kernel/signal.c	old_sigset_t *	-	-	-	-
74	sys_sethostname	kernel/sys.c	char *	int	-	-	-
75	sys_setrlimit	kernel/sys.c	unsigned int	struct rlimit *	-	-	-
76	sys_getrlimit	kernel/sys.c	unsigned int	struct rlimit *	-	-	-
77	sys_getrusage	kernel/sys.c	int	struct rusage *	-	-	-
78	sys_gettimeofday	kernel/time.c	struct timeval *	struct timezone *	-	-	-
79	sys_settimeofday	kernel/time.c	struct timeval *	struct timezone *	-	-	-
80	sys_getgroups	kernel/sys.c	int	gid_t *	-	-	-
81	sys_setgroups	kernel/sys.c	int	gid_t *	-	-	-
82	old_select	arch/i386/kernel/sys_i386.c	struct sel_arg_struct *	-	-	-	-
83	sys_symlink	fs/namei.c	const char *	const char *	-	-	-
84	sys_lstat	fs/stat.c	char *	struct __old_kernel_stat *	-	-	-
85	sys_readlink	fs/stat.c	const char *	char *	int	-	-
86	sys_uselib	fs/exec.c	const char *	-	-	-	-
87	sys_swapon	mm/swapfile.c	const char *	int	-	-	-
88	sys_reboot	kernel/sys.c	int	int	int	void *	-
89	old_readdir	fs/readdir.c	unsigned int	void *	unsigned int	-	-
90	old_mmap	arch/i386/kernel/sys_i386.c	struct mmap_arg_struct *	-	-	-	-
91	sys_munmap	mm/mmap.c	unsigned long	size_t	-	-	-
92	sys_truncate	fs/open.c	const char *	unsigned long	-	-	-
93	sys_ftruncate	fs/open.c	unsigned int	unsigned long	-	-	-

94	sys_fchmod	fs/open.c	unsigned int	mode_t	-	-	-
95	sys_fchown	fs/open.c	unsigned int	uid_t	gid_t	-	-
96	sys_getpriority	kernel/sys.c	int	int	-	-	-
97	sys_setpriority	kernel/sys.c	int	int	int	-	-
99	sys_statfs	fs/open.c	const char *	struct statfs *	-	-	-
100	sys_fstatfs	fs/open.c	unsigned int	struct statfs *	-	-	-
101	sys_ioperm	arch/i386/kernel/ioport.c	unsigned long	unsigned long	int	-	-
102	sys_socketcall	net/socket.c	int	unsigned long *	-	-	-
103	sys_syslog	kernel/printk.c	int	char *	int	-	-
104	sys_setitimer	kernel/itimer.c	int	struct itimerval *	struct itimerval *	-	-
105	sys_getitimer	kernel/itimer.c	int	struct itimerval *	-	-	-
106	sys_newstat	fs/stat.c	char *	struct stat *	-	-	-
107	sys_newlstat	fs/stat.c	char *	struct stat *	-	-	-
108	sys_newfstat	fs/stat.c	unsigned int	struct stat *	-	-	-
109	sys_uname	arch/i386/kernel/sys_i386.c	struct old_utsname *	-	-	-	-
110	sys_iopl	arch/i386/kernel/ioport.c	unsigned long	-	-	-	-
111	sys_vhangup	fs/open.c	-	-	-	-	-
112	sys_idle	arch/i386/kernel/process.c	-	-	-	-	-
113	sys_vm86old	arch/i386/kernel/vm86.c	unsigned long	struct vm86plus_struct *	-	-	-
114	sys_wait4	kernel/exit.c	pid_t	unsigned long *	int options	struct rusage *	-
115	sys_swapoff	mm/swapfile.c	const char *	-	-	-	-
116	sys_sysinfo	kernel/info.c	struct sysinfo *	-	-	-	-
117	sys_ipc(*Note)	arch/i386/kernel/sys_i386.c	uint	int	int	int	void *
118	sys_fsync	fs/buffer.c	unsigned int	-	-	-	-
119	sys_sigreturn	arch/i386/kernel/signal.c	unsigned long	-	-	-	-
120	sys_clone	arch/i386/kernel/process.c	struct pt_regs	-	-	-	-
121	sys_setdomainname	kernel/sys.c	char *	int	-	-	-
122	sys_newuname	kernel/sys.c	struct new_utsname *	-	-	-	-
123	sys_modify_ldt	arch/i386/kernel/ldt.c	int	void *	unsigned long	-	-
124	sys_adjtimex	kernel/time.c	struct timex *	-	-	-	-
125	sys_mprotect	mm/mprotect.c	unsigned long	size_t	unsigned long	-	-
126	sys_sigprocmask	kernel/signal.c	int	old_sigset_t *	old_sigset_t *	-	-
127	sys_create_module	kernel/module.c	const char *	size_t	-	-	-
128	sys_init_module	kernel/module.c	const char *	struct module *	-	-	-
129	sys_delete_module	kernel/module.c	const char *	-	-	-	-
130	sys_get_kernel_syms	kernel/module.c	struct kernel_sym *	-	-	-	-
131	sys_quotactl	fs/dquot.c	int	const char *	int	caddr_t	-
132	sys_getpgid	kernel/sys.c	pid_t	-	-	-	-
133	sys_fchdir	fs/open.c	unsigned int	-	-	-	-
134	sys_bdflush	fs/buffer.c	int	long	-	-	-
135	sys_sysfs	fs/super.c	int	unsigned long	unsigned long	-	-
136	sys_personality	kernel/exec_domain.c	unsigned long	-	-	-	-
138	sys_setfsuid	kernel/sys.c	uid_t	-	-	-	-
139	sys_setfsgid	kernel/sys.c	gid_t	-	-	-	-
					unsigned		unsigned

140	sys_llseek	fs/read_write.c	unsigned int	unsigned long	long	loff_t *	int
141	sys_getdents	fs/readdir.c	unsigned int	void *	unsigned int	-	-
142	sys_select	fs/select.c	int	fd_set *	fd_set *	fd_set *	struct timeval *
143	sys_flock	fs/locks.c	unsigned int	unsigned int	-	-	-
144	sys_msync	mm/filemap.c	unsigned long	size_t	int	-	-
145	sys_readv	fs/read_write.c	unsigned long	const struct iovec *	unsigned long	-	-
146	sys_writev	fs/read_write.c	unsigned long	const struct iovec *	unsigned long	-	-
147	sys_getsid	kernel/sys.c	pid_t	-	-	-	-
148	sys_fdatasync	fs/buffer.c	unsigned int	-	-	-	-
149	sys_sysctl	kernel/sysctl.c	struct __sysctl_args *	-	-	-	-
150	sys_mlock	mm/mlock.c	unsigned long	size_t	-	-	-
151	sys_munlock	mm/mlock.c	unsigned long	size_t	-	-	-
152	sys_mlockall	mm/mlock.c	int	-	-	-	-
153	sys_munlockall	mm/mlock.c	-	-	-	-	-
154	sys_sched_setparam	kernel/sched.c	pid_t	struct sched_param *	-	-	-
155	sys_sched_getparam	kernel/sched.c	pid_t	struct sched_param *	-	-	-
156	sys_sched_setscheduler	kernel/sched.c	pid_t	int	struct sched_param *	-	-
157	sys_sched_getscheduler	kernel/sched.c	pid_t	-	-	-	-
158	sys_sched_yield	kernel/sched.c	-	-	-	-	-
159	sys_sched_get_priority_max	kernel/sched.c	int	-	-	-	-
160	sys_sched_get_priority_min	kernel/sched.c	int	-	-	-	-
161	sys_sched_rr_get_interval	kernel/sched.c	pid_t	struct timespec *	-	-	-
162	sys_nanosleep	kernel/sched.c	struct timespec *	struct timespec *	-	-	-
163	sys_mremap	mm/mremap.c	unsigned long	unsigned long	unsigned long	unsigned long	-
164	sys_setresuid	kernel/sys.c	uid_t	uid_t	uid_t	-	-
165	sys_getresuid	kernel/sys.c	uid_t *	uid_t *	uid_t *	-	-
166	sys_vm86	arch/i386/kernel/vm86.c	struct vm86_struct *	-	-	-	-
167	sys_query_module	kernel/module.c	const char *	int	char *	size_t	size_t *
168	sys_poll	fs/select.c	struct pollfd *	unsigned int	long	-	-
169	sys_nfsservctl	fs/filesystems.c	int	void *	void *	-	-
170	sys_setresgid	kernel/sys.c	gid_t	gid_t	gid_t	-	-
171	sys_getresgid	kernel/sys.c	gid_t *	gid_t *	gid_t *	-	-
172	sys_prctl	kernel/sys.c	int	unsigned long	unsigned long	unsigned long	unsigned long
173	sys_rt_sigreturn	arch/i386/kernel/signal.c	unsigned long	-	-	-	-
174	sys_rt_sigaction	kernel/signal.c	int	const struct sigaction *	struct sigaction *	size_t	-
175	sys_rt_sigprocmask	kernel/signal.c	int	sigset_t *	sigset_t *	size_t	-
176	sys_rt_sigpending	kernel/signal.c	sigset_t *	size_t	-	-	-
177	sys_rt_sigtimedwait	kernel/signal.c	const sigset_t *	siginfo_t *	const struct timespec *	size_t	-
178	sys_rt_sigqueueinfo	kernel/signal.c	int	int	siginfo_t *	-	-
179	sys_rt_sigsuspend	arch/i386/kernel/signal.c	sigset_t *	size_t	-	-	-
180	sys_pread	fs/read_write.c	unsigned int	char *	size_t	loff_t	-
181	sys_pwrite	fs/read_write.c	unsigned int	const char *	size_t	loff_t	-

182	sys_chown	fs/open.c	const char *	uid_t	gid_t	-	-
183	sys_getcwd	fs/dcache.c	char *	unsigned long	-	-	-
184	sys_capget	kernel/capability.c	cap_user_header_t	cap_user_data_t	-	-	-
185	sys_capset	kernel/capability.c	cap_user_header_t	const cap_user_data_t	-	-	-
186	sys_sigaltstack	arch/i386/kernel/signal.c	const stack_t *	stack_t *	-	-	-
187	sys_sendfile	mm/filemap.c	int	int	off_t *	size_t	-
190	sys_vfork	arch/i386/kernel/process.c	struct pt_regs	-	-	-	-

Note for sys_ipc (117): this syscall takes six arguments, so it can't fit into the five registers %ebx - %edi; the last parameter (not shown) is of type 'long'. This syscall requires a special call method where a pointer is put in %ebx which points to an array containing the six arguments.

System Call Numbers

For the numbers of the syscalls, look in arch/i386/kernel/entry.S for **sys_call_table**. The syscall numbers are offsets into that table. Several spots in the table are occupied by the syscall **sys_ni_syscall**. This is a placeholder that either replaces an obsolete syscall or reserves a spot for future syscalls.

Incidentally, the **system calls** are called from the function **system_call** in the same file; in particular, they are called with the assembly instruction 'call *SYMBOL_NAME(sys_call_table)(,%eax,4)'. The part '*SYMBOL_NAME(sys_call_table)' just gets replaced by a symbol name in **sys_call_table**. **SYMBOL_NAME** is a macro defined in include/**linux**/linkage.h, and it just replaces itself with its argument.

Typedefs

Here are the typedef declarations in the prototypes above:

atomic_t	include/asm/atomic.h: #ifdef __SMP__ typedef struct { volatile int counter; } atomic_t; #else typedef struct { int counter; } atomic_t; #endif
caddr_t	include/asm/posix_types.h:typedef char * __kernel_caddr_t; include/**linux**/types.h:typedef __kernel_caddr_t caddr_t;

cap_user_header_t	include/**linux**/capability.h:
	typedef struct __user_cap_header_struct {
	  __u32 version;
	  int pid;
	} *cap_user_header_t;

cap_user_data_t	include/**linux**/capability.h:
	typedef struct __user_cap_data_struct {
	  __u32 effective;
	  __u32 permitted;
	  __u32 inheritable;
	} *cap_user_data_t;

clock_t	include/asm/posix_types.h:typedef long __kernel_clock_t; include/**linux**/types.h:typedef __kernel_clock_t clock_t;
dev_t	include/asm/posix_types.h:typedef unsigned short __kernel_dev_t; include/**linux**/types.h:typedef __kernel_dev_t dev_t;
fd_set	include/**linux**/posix_types.h #define __FD_SETSIZE 1024 #define __NFDBITS (8 * sizeof(unsigned long)) #define __FDSET_LONGS (__FD_SETSIZE/__NFDBITS) (==> __FDSET_LONGS == 32) typedef struct { unsigned long fds_bits [__FDSET_LONGS]; } __kernel_fd_set; include/**linux**/types.h:typedef __kernel_fd_set fd_set;
gid_t	include/asm/posix_types.h:typedef unsigned short __kernel_gid_t; include/**linux**/types.h:typedef __kernel_gid_t gid_t;
__kernel_daddr_t	include/asm/posix_types.h:typedef int __kernel_daddr_t;

__kernel_fsid_t	include/asm/posix_types.h: typedef struct { int __val[2]; } __kernel_fsid_t;
__kernel_ino_t	include/asm/posix_types.h:typedef unsigned long __kernel_ino_t;
__kernel_size_t	include/asm/posix_types.h:typedef unsigned int __kernel_size_t;
loff_t	include/asm/posix_types.h:typedef long long __kernel_loff_t; include/**linux**/types.h:typedef __kernel_loff_t loff_t;
mode_t	include/asm/posix_types.h:typedef unsigned short __kernel_mode_t; include/**linux**/types.h:typedef __kernel_mode_t mode_t;
off_t	include/asm/posix_types.h:typedef long __kernel_off_t; include/**linux**/types.h:typedef __kernel_off_t off_t;
old_sigset_t	include/asm/signal.h:typedef unsigned long old_sigset_t;
pid_t	include/asm/posix_types.h:typedef int __kernel_pid_t; include/**linux**/types.h:typedef __kernel_pid_t pid_t;
__sighandler_t	include/asm/signal.h:typedef void (*__sighandler_t)(int);

siginfo_t	include/asm/siginfo.h:
	#define SI_MAX_SIZE 128
	#define SI_PAD_SIZE ((SI_MAX_SIZE/sizeof(int)) - 3)
	(==> SI_PAD_SIZE == 29)
	
	typedef struct siginfo {
	  int si_signo;
	  int si_errno;
	  int si_code;
	
	  union {
	  int _pad[SI_PAD_SIZE];
	
	  /* kill() */
	  struct {
	  pid_t _pid; /* sender's pid */
	  uid_t _uid; /* sender's uid */
	  } _kill;
	
	  /* POSIX.1b timers */
	  struct {
	  unsigned int _timer1;
	  unsigned int _timer2;
	  } _timer;
	
	  /* POSIX.1b signals */
	  struct {
	  pid_t _pid; /* sender's pid */
	  uid_t _uid; /* sender's uid */
	  sigval_t _sigval;
	  } _rt;
	
	  /* SIGCHLD */
	  struct {
	  pid_t _pid; /* which child */
	  uid_t _uid; /* sender's uid */
	  int _status; /* exit code */
	  clock_t _utime;
	  clock_t _stime;
	  } _sigchld;
	
	  /* SIGILL, SIGFPE, SIGSEGV, SIGBUS */
	  struct {
	  void *_addr; /* faulting insn/memory ref. */
	  } _sigfault;
	
	  /* SIGPOLL */
	  struct {
	  int _band; /* POLL_IN, POLL_OUT, POLL_MSG */
	  int _fd;
	  } _sigpoll;
	  } _sifields;

	} siginfo_t;
sigset_t	include/asm/signal.h:typedef unsigned long sigset_t;
size_t	include/asm/posix_types.h:typedef unsigned int __kernel_size_t; include/**linux**/types.h:typedef __kernel_size_t size_t;
ssize_t	include/asm/posix_types.h:typedef int __kernel_ssize_t; include/**linux**/types.h:typedef __kernel_ssize_t ssize_t;
stack_t	include/asm/signal.h: typedef struct sigaltstack { void *ss_sp; int ss_flags; size_t ss_size; } stack_t;
suseconds_t	include/asm/posix_types.h:typedef long __kernel_suseconds_t; include/**linux**/types.h:typedef __kernel_suseconds_t suseconds_t;
time_t	include/asm/posix_types.h:typedef long __kernel_time_t; include/**linux**/types.h:typedef __kernel_time_t time_t;
uid_t	include/asm/posix_types.h:typedef unsigned short __kernel_uid_t; include/**linux**/types.h:typedef __kernel_uid_t uid_t;
uint	include/**linux**/types.h:typedef unsigned int uint;
__u32	include/asm/types.h:typedef unsigned int __u32;

Structs

Here are the struct declarations for the table at the top:

exception_table_entry	include/**linux**/module.h:
	struct exception_table_entry {
	unsigned long insn, fixup;
	};

iovec	include/**linux**/uio.h: struct iovec { void *iov_base; __kernel_size_t iov_len; };

itimerval	include/**linux**/time.h:
	struct itimerval {
	struct timeval it_interval; /* timer interval */
	struct timeval it_value; /* current value */
	};

kernel_sym	include/**linux**/module.h:
	struct kernel_sym {
	unsigned long value;
	char name[60];
	};

mmap_arg_struct	arch/i386/kernel/sys_i386.c:
	struct mmap_arg_struct {
	unsigned long addr;
	unsigned long len;
	unsigned long prot;
	unsigned long flags;
	unsigned long fd;
	unsigned long offset;
	};

module	include/**linux**/module.h:
	struct module {
	unsigned long size_of_struct; /* sizeof(module) */
	struct module *next;
	const char *name;
	unsigned long size;
	union {
	atomic_t usecount;
	long pad;
	} uc;
	unsigned long flags; /* AUTOCLEAN et al */

	unsigned nsyms; unsigned ndeps; struct module_symbol *syms; struct module_ref *deps; struct module_ref *refs; int (*init)(void); void (*cleanup)(void); const struct exception_table_entry *ex_table_start; const struct exception_table_entry *ex_table_end; /* Members past this point are extensions to the basic module support and are optional. Use mod_opt_member() to examine them. */ const struct module_persist *persist_start; const struct module_persist *persist_end; int (*can_unload)(void); };
module_persist	include/**linux**/module.h: struct module_persist; /* yes, it's empty */

module_ref	include/**linux**/module.h:
	struct module_ref {
	struct module *dep; /* "parent" pointer */
	struct module *ref; /* "child" pointer */
	struct module_ref *next_ref;
	};

module_symbol	include/**linux**/module.h:
	struct module_symbol {
	unsigned long value;
	const char *name;
	};

new_utsname	include/**linux**/utsname.h:
	struct new_utsname {
	char sysname[65];
	char nodename[65];
	char release[65];
	char version[65];
	char machine[65];
	char domainname[65];
	};

__old_kernel_stat	include/asm/stat.h:
	struct __old_kernel_stat {
	unsigned short st_dev;
	unsigned short st_ino;
	unsigned short st_mode;
	unsigned short st_nlink;
	unsigned short st_uid;
	unsigned short st_gid;
	unsigned short st_rdev;
	unsigned long st_size;
	unsigned long st_atime;
	unsigned long st_mtime;
	unsigned long st_ctime;
	};

oldold_utsname	include/**linux**/utsname.h:
	struct oldold_utsname {
	char sysname[9];
	char nodename[9];
	char release[9];
	char version[9];
	char machine[9];
	};

old_sigaction	include/asm/signal.h:
	struct old_sigaction {
	__sighandler_t sa_handler;
	old_sigset_t sa_mask;
	unsigned long sa_flags;
	void (*sa_restorer)(void);
	};

old_utsname	include/**linux**/utsname.h:
	struct old_utsname {
	char sysname[65];
	char nodename[65];
	char release[65];
	char version[65];
	char machine[65];
	};

pollfd	include/asm/poll.h:
	struct pollfd {
	int fd;
	short events;
	short revents;
	};

pt_regs	include/asm/ptrace.h:
	struct pt_regs {
	long ebx;
	long ecx;
	long edx;
	long esi;
	long edi;
	long ebp;
	long eax;
	int xds;
	int xes;
	long orig_eax;
	long eip;
	int xcs;
	long eflags;
	long esp;
	int xss;
	};

revectored_struct	include/asm/vm86.h:
	struct revectored_struct {
	unsigned long __map[8];
	};

rlimit	include/**linux**/resource.h:
	struct rlimit {
	long rlim_cur;
	long rlim_max;
	};

rusage	include/**linux**/resource.h:
	struct rusage {
	struct timeval ru_utime; /* user time used */
	struct timeval ru_stime; /* **system** time used */
	long ru_maxrss; /* maximum resident set size */
	long ru_ixrss; /* integral shared memory size */
	long ru_idrss; /* integral unshared data size */
	long ru_isrss; /* integral unshared stack size */
	long ru_minflt; /* page reclaims */
	long ru_majflt; /* page faults */
	long ru_nswap; /* swaps */
	long ru_inblock; /* block input operations */
	long ru_oublock; /* block output operations */
	long ru_msgsnd; /* messages sent */
	long ru_msgrcv; /* messages received */
	long ru_nsignals; /* signals received */
	long ru_nvcsw; /* voluntary context switches */
	long ru_nivcsw; /* involuntary " */
	};

sched_param	include/**linux**/sched.h:
	struct sched_param {
	int sched_priority;
	};

sel_arg_struct	arch/i386/kernel/sys_i386.c: struct sel_arg_struct { unsigned long n; fd_set *inp, *outp, *exp; struct timeval *tvp;

	};

sigaction	include/asm/signal.h:
	struct sigaction {
	  __sighandler_t sa_handler;
	  unsigned long sa_flags;
	  void (*sa_restorer)(void);
	  sigset_t sa_mask; /* mask last for extensibility */
	};

stat	include/asm/stat.h:
	struct stat {
	  unsigned short st_dev;
	  unsigned short __pad1;
	  unsigned long st_ino;
	  unsigned short st_mode;
	  unsigned short st_nlink;
	  unsigned short st_uid;
	  unsigned short st_gid;
	  unsigned short st_rdev;
	  unsigned short __pad2;
	  unsigned long st_size;
	  unsigned long st_blksize;
	  unsigned long st_blocks;
	  unsigned long st_atime;
	  unsigned long __unused1;
	  unsigned long st_mtime;
	  unsigned long __unused2;
	  unsigned long st_ctime;
	  unsigned long __unused3;
	  unsigned long __unused4;
	  unsigned long __unused5;

statfs	include/asm/statfs.h:
	struct statfs {
	  long f_type;
	  long f_bsize;
	  long f_blocks;
	  long f_bfree;
	  long f_bavail;
	  long f_files;
	  long f_ffree;
	  __kernel_fsid_t f_fsid;
	  long f_namelen;
	  long f_spare[6];
	};

__sysctl_args	include/**linux**/sysctl.h
	struct __sysctl_args {
	  int *name;
	  int nlen;
	  void *oldval;
	  size_t *oldlenp;
	  void *newval;
	  size_t newlen;
	  unsigned long __unused[4];
	};

sysinfo	include/**linux**/kernel.h:
	struct sysinfo {
	  long uptime; /* Seconds since boot */
	  unsigned long loads[3]; /* 1, 5, and 15 minute load averages */
	  unsigned long totalram; /* Total usable main memory size */
	  unsigned long freeram; /* Available memory size */
	  unsigned long sharedram; /* Amount of shared memory */
	  unsigned long bufferram; /* Memory used by buffers */
	  unsigned long totalswap; /* Total swap space size */
	  unsigned long freeswap; /* swap space still available */
	  unsigned short procs; /* Number of current processes */
	  char _f[22]; /* Pads structure to 64 bytes */
	};

timex	include/**linux**/timex.h: struct timex { unsigned int modes; /* mode selector */

	long offset; /* time offset (usec) */ long freq; /* frequency offset (scaled ppm) */ long maxerror; /* maximum error (usec) */ long esterror; /* estimated error (usec) */ int status; /* clock command/status */ long constant; /* pll time constant */ long precision; /* clock precision (usec) (read only) */ long tolerance; /* clock frequency tolerance (ppm) * (read only) */ struct timeval time; /* (read only) */ long tick; /* (modified) usecs between clock ticks */ long ppsfreq; /* pps frequency (scaled ppm) (ro) */ long jitter; /* pps jitter (us) (ro) */ int shift; /* interval duration (s) (shift) (ro) */ long stabil; /* pps stability (scaled ppm) (ro) */ long jitcnt; /* jitter limit exceeded (ro) */ long calcnt; /* calibration intervals (ro) */ long errcnt; /* calibration errors (ro) */ long stbcnt; /* stability limit exceeded (ro) */ int :32; int :32; int :32; int :32; int :32; int :32; int :32; int :32; int :32; int :32; int :32; int :32; };

timespec	include/**linux**/time.h:
	struct timespec {
	time_t tv_sec; /* seconds */
	long tv_nsec; /* nanoseconds */
	};

timeval	include/**linux**/time.h:
	struct timeval {
	time_t tv_sec; /* seconds */
	suseconds_t tv_usec; /* microseconds */
	};

timezone	include/**linux**/time.h:
	struct timezone {
	int tz_minuteswest; /* minutes west of Greenwich */
	int tz_dsttime; /* type of dst correction */
	};

tms	include/**linux**/times.h
	struct tms {
	clock_t tms_utime;
	clock_t tms_stime;
	clock_t tms_cutime;
	clock_t tms_cstime;
	};

ustat	include/**linux**/types.h:
	struct ustat {
	__kernel_daddr_t f_tfree;
	__kernel_ino_t f_tinode;
	char f_fname[6];
	char f_fpack[6];
	};

utimbuf	include/**linux**/utime.h:
	struct utimbuf {
	time_t actime;
	time_t modtime;
	};

vm86plus_info_struct	include/asm/vm86.h:
	struct vm86plus_info_struct {
	unsigned long force_return_for_pic:1;
	unsigned long vm86dbg_active:1;
	unsigned long vm86dbg_TFpendig:1;
	unsigned long unused:28;
	unsigned long is_vm86pus:1;
	unsigned char vm86dbg_intxxtab[32];
	};

vm86plus_struct	include/asm/vm86.h:
	struct vm86plus_struct {
	  struct vm86_regs regs;
	  unsigned long flags;
	  unsigned long screen_bitmap;
	  unsigned long cpu_type;
	  struct revectored_struct int_revectored;
	  struct revectored_struct int21_revectored;
	  struct vm86plus_info_struct vm86plus;
	};

vm86_regs	include/asm/vm86.h:
	struct vm86_regs {
	/* normal regs, with special meaning for the segment descriptors.. */
	  long ebx;
	  long ecx;
	  long edx;
	  long esi;
	  long edi;
	  long ebp;
	  long eax;
	  long __null_ds;
	  long __null_es;
	  long __null_fs;
	  long __null_gs;
	  long orig_eax;
	  long eip;
	  unsigned short cs, __csh;
	  long eflags;
	  long esp;
	  unsigned short ss, __ssh;
	/* these are specific to v86 mode: */
	  unsigned short es, __esh;
	  unsigned short ds, __dsh;
	  unsigned short fs, __fsh;
	  unsigned short gs, __gsh;
	};

vm86_struct	include/asm/vm86.h:
	struct vm86_struct {
	  struct vm86_regs regs;
	  unsigned long flags;
	  unsigned long screen_bitmap;
	  unsigned long cpu_type;
	  struct revectored_struct int_revectored;
	  struct revectored_struct int21_revectored;
	};

Arquitectura de Computadoras
TP3 - Assembler y C

Introducción **2**

Llamadas de Assembler a C **2**

Ejemplo 1 2
Pregunta 3
Ejemplo 2 3
Preguntas 5

Registros a preservar entre llamadas **5**

Stack Frame **5**

Ejercicios **6**

Ejercicio 1 6

Alineamiamiento a palabra **6**

Responder 7
Ejercicio 2 7

Llamadas de C a Assembler **8**

Ejemplo 3 8

Recepción de parámetros **9**

Ejercicio 3 10

Variables **10**

Ejemplo 4 10
Reponder 11
Ejemplo 5 11

Ejercicios **11**

Ejercicio 5 11
Ejercicio 6 11
Ejercicio 7 12
Ejercicio 8 13
Ejercicio 9 13
Ejercicio 10 13
Ejercicio 11 13
Ejercicio 12 13
Ejercicio 13 15

x86_64 ABI **15**

Clasificación de argumentos 15
Pasaje de argumentos 15
Retorno de valores 15
Registros a preservar 16

Ejercicios **17**
Consigna 17
Ejercicios de debugging 17
Ejercicio 14 17
Ejercicio 15 18
Ejercicio 16 18
Ejercicio 17 19
Ejercicio 18 20

Introducción

Todo programa que corra en una computadora, tiene que estar en un lenguaje que ésta entienda; en código de máquina. El código de máquina hace uso de los registros del procesador, del stack, de las interrupciones, etc. Así que un programa escrito en C, en Assembler, COBOL, Pascal, Ada, etc; a fin de cuentas termina en código de máquina, lo que implica es que siempre se puede hacer una interface para conectar los lenguajes entre sí. En particular, cómo entre código de máquina y Assembler hay una relación de biyectiva, es fácil conectar este lenguaje con cualquier otro, ya que con Assembler hay que respetar el funcionamiento interno del otro lenguaje para poder interactuar.

Es interesante poder hacer una interface con Assembler ya que éste tiene control directo sobre las operaciones del procesador. No como los lenguajes de propósito general que se abstraen de la arquitectura. Con este lenguaje se pueden deshabilitar las interrupciones, habilitar modo protegido, cambiar la dirección del directorio de páginas, modificar el stack pointer, etc.

El lenguaje C es por excelencia el lenguaje de elección para el desarrollo básico de un sistema operativo. Por su bajo nivel, versatilidad, manejo de memoria y su independencia de servicios que necesitan otros lenguajes. Otros lenguajes de programación de más alto nivel, por ejemplo C++, necesitan configuraciones especiales para funcionar.

La convención de cómo realizar las llamadas entre distintos lenguajes, cómo se reserva espacio para las variables, cómo se alinean los datos, etc; es parte de la **Application Binary Interface (ABI)**.

Llamadas de Assembler a C

Este camino es el más fácil de entender. Si recordamos que al final de cuentas, una función en C es una dirección de memoria, no debería ser difícil hacer una llamada desde ASM a C. Vamos a utilizar el programa nasm para compilar los archivos en asm, gcc para compilar los archivos en C y también para linkeditarlos entre sí y contra la biblioteca estándar de C.

Ejemplo 1

Escribir una función en C que imprima por pantalla “Hola Mundo”. Luego, escribir una función en ASM que llame a dicha función

```
;main.asm
GLOBAL main
```

```
EXTERN hello_world

section .text

main:
      call hello_world

      mov eax, 1  ;systemcall exit()
      mov ebx, 0  ;parametro para exit()
      int 80h
```

```
//hello.c
#include <stdio.h>

void hello_world();

void hello_world() {
      printf ("Hello World!\n");
}
```

Para compilar y correr los archivos anteriores correr las siguientes líneas:

```
$> nasm -f elf32 main.asm
$> gcc -c -m32 hello.c
$> gcc -m32 main.o hello.o -o hello
$> ./hello
```

La directiva **-f elf32** le indica a nasm que genere un archivo objeto con formato elf (executable linux format) para una arquitectura de 32 bits. Algo equivalente hace la directiva **-m32**. Si no se especifica esto, el formato default es el de la máquina actual.

Note que ahora, en lugar de utilizar la etiqueta _start ahora se utiliza la etiqueta main. Esto es porque la biblioteca de C ya tiene una etiqueta de entrada, que configura el sistema para luego llamar a nuestro main.

Pregunta

Ya que ahora estamos en un entorno de C, ¿De qué otra forma se podría haber terminado el programa?

Ejemplo 2

En este ejemplo, vamos a pasar un valor a su representación en ASCII numérica. Recordar cómo es el pasaje de pasaje de parámetros en C, por Stack en 32 bits:

Si una función tiene la siguiente forma:

```
int fnc(arg1, arg2, arg3);
```

Entonces, los argumentos se pasan de derecha a izquierda, de tal forma, que el primer argumento, sea el primero en el stack. Es decir, primero **arg3**, luego **arg2** y por último **arg1**. Luego, el resultado de dicha función se devuelve por el registro eax.

La función de C sprintf escribe un string con formato en un stream, tiene el siguiente prototipo:

```
int sprintf(char *str, const char *fmt, ...);
```

Esto quiere decir que para llamar a esta función, primero será necesario pushear al stack los argumentos variables, luego la cadena de formato (que le indicará a la función cómo levantar la información) y por último el destino. Luego, se llama a la función puts que imprime por pantalla una cadena.

```
int puts(const char *str);
```

Código:

```
;main.asm
GLOBAL main
EXTERN puts, sprintf

section .rodata
fmt db “%d”, 0
number dd 123456790

section .text
main:
      push dword [number]
      push fmt
      push buffer

      call sprintf

      add esp, 3*4

      push buffer
      call puts

      add esp, 4
      ret

section .bss
buffer resb 40
```

Para compilar este código, simplemente hay que correr las siguientes líneas:

```
$> nasm -f elf32 main.asm
$> gcc -m32 main.o -o main
$> ./main
```

Preguntas

1. ¿Porqué se le suma al stack, luego de llamar a la función sprintf, 12 bytes? ¿Porqué se hace lo mismo luego de llamar a puts?
2. ¿Cuál es el valor de retorno de la función main?

Registros a preservar entre llamadas

Entre llamadas de C, hay ciertos registros que las funciones deben preservar para no afectar el funcionamiento de otras funciones. Las funciones que llaman deben tener en cuenta que registros pueden cambiar, de tal forma de hacer un backup antes de llamarlas.

Los registros a preservar entre las llamadas de las funciones son:

- ebx
- esi
- edi
- ebp
- esp

Es decir, cuando una función termina, debe dejar exactamente los registros anteriores como los recibió.

Stack Frame

Es parte de la ABI de C, cuando se entra a una función, armar esta estructura. Su función es preservar los registros de stack y proveer un sistema estándar de acceso a los parámetros de dicha función.

Con este sistema, la el stack queda de la siguiente forma:

EBP	EBP del contexto anterior
EBP + 4	Dirección de Retorno
EBP + 8	Primer argumento
EBP + 12	Segundo argumento
EBP + (n+1)*4	n-esimo argumento

En C main es una función que tiene el siguiente prototipo

```
int main(int argc, char *argv[]);
```

El program loader, además de configurar algunas cosas, pasa como parámetro de dicha función los argumentos del programa. Entonces se pueden acceder de la siguiente forma:

```
;main.asm
GLOBAL main
EXTERN printf

```

```
section .rodata
fmt db “Cantidad de argumentos: %d\n”, 0

section .text
main:
      push ebp          ;Armado de stack frame
      mov ebp, esp      ;

      push dword [ebp+8]
      push fmt
      call printf
      add esp, 2*4

      mov eax, 0

      mov esp, ebp      ;Desarmado de stack frame
      pop ebp           ;
      ret
```

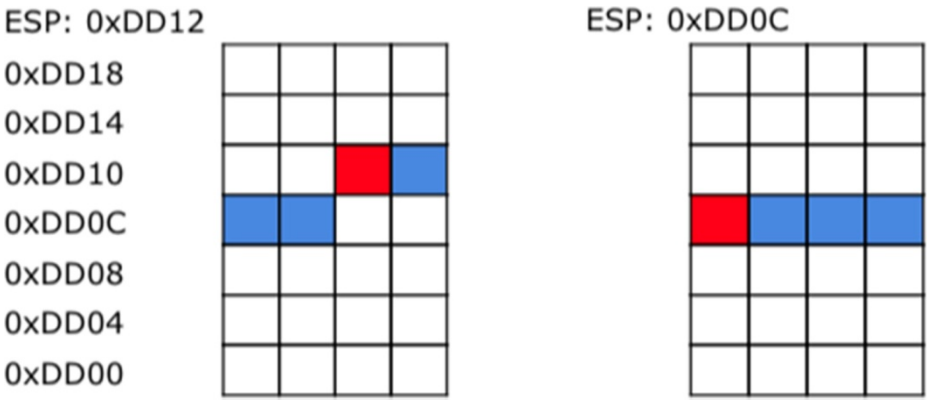
Ejercicios

Ejercicio 1

Modificar el ejemplo anterior, para que además de imprimir la cantidad de argumentos, imprima cada uno de los argumentos.

Alineamiamiento a palabra

El procesador cada vez que accede a memoria, lo hace leyendo y escribiendo siempre 4 bytes (el tamaño de la palabra en una arquitectura x86).



Cuando el procesador accede a un dato des-alineado, debe hacer dos lecturas, una para pedir la primer porción de los datos, y otra para la segunda porción; es decir, dos accesos. Para la escritura requiere más operaciones, ya que debe leer de memoria para salvar el dato que no va a sobrescribir, luego combinarlo con el dato que si va a escribir para la primera porción, y luego recuperar el dato de la segunda porción para escribir la segunda porción.

Una forma muy fácil de arreglar este problema, es negar los últimos 4 bits. Cómo el stack crece hacia menores valores de esp, seguro esa posición va a estar por detrás del puntero actual. De esta forma se puede hacer un acceso a memoria más eficiente.

```
GLOBAL main

ALIGN 4

main:
      push ebp
      mov ebp, esp

      ;declaración de variables

      and esp, -16

      ;... programa

      mov esp, ebp
      pop ebp
      ret
```

Responder

¿Cual es la representación de -16 en hexadecimal?

Ejercicio 2

Escriba un programa en C, que realice lo mismo que el ejercicio anterior (imprimir por salida estándar la cantidad de argumentos del programa),

```
//arguments.c
#include <stdio.h>

int main(int argc, char *argv[]) {
      printf("Cantidad de argumentos %d\n", argc);
      return 0;
}
```

Compilarlo con el siguiente comando (que elimina un montón de elementos de control de gcc):

```
$> gcc -c arguments.c -m32 -fno-dwarf2-cfi-asm -fno-exceptions -S -fno-asynchronous-unwind-tables -masm=intel
```

Compare la salida del archivo arguments.s con el ejemplo provisto en ASM. Más allá de los elementos de control, ¿Qué diferencia encuentra? ¿Qué similitudes?

Para pensar: ¿Se imagina porque no se deben comparar las estructuras por valor, sino miembro por miembro? ¿Porqué habría basura entre sus componentes?

Llamadas de C a Assembler

Este es el camino que tiene una aplicación más práctica. ¿Qué punto tiene programar en Assembler cuando se tiene un lenguaje eficiente, práctico y liviano? Salvo que exista una razón muy puntual; cómo que no exista un compilador; evitaremos programar en Assembler.

Nuevamente, las funciones a fin de cuentas son direcciones, así que es posible llamar funciones desde C hacia Assembler.

Ejemplo 3

Si no contamos con una biblioteca estándar que acceda a la API del SO para enviar o recibir información, debería ser implementado por nosotros. Ya vimos cómo comunicarnos con el SO completamente en Assembler, ahora tendremos que hacer nosotros una función auxiliar para comunicarnos a través de C.

El PID (Process ID) es un número que utiliza Linux para identificar los procesos que están corriendo. Si se corre el comando **ps x**, se puede ver una lista de los procesos que están corriendo actualmente que son del usuario. Obviamente, como **ps** es un proceso, también aparece en la lista.

No hay ninguna función estándar que informe cual es el **pid** de un programa. Cómo es información que maneja el OS, habrá que pedírsela a él.

```
//main.c
#include<stdio.h>

unsigned int pid();

int main(int argc, char *argv[]) {
      int mpid = pid();
      printf("Process Id: %d\n", mpid);
      return 0;
}
```

```
; libasm.asm
GLOBAL pid

pid:
      push ebp
      mov ebp, esp

      mov eax, 0x14     ;syscall getpid
      int 0x80
      ;el resultado ya está en eax

      mov esp, ebp
      pop ebp
      ret
```

Compilar y correr el código de ejemplo con las siguientes líneas:

```
$> nasm -f elf32 libasm.asm
$> gcc -c -m32 main.c
$> gcc -m32 main.o libasm.o -o pid
$> ./pid
```

Ejecútelo varias veces para comprobar que el **pid** va en aumento.

Compile el archivo main.c de tal forma de obtener su salida en Assembler, compruebe que efectivamente se hace una llamada a la función **pid()** mediante la línea.

```
call pid
```

Para que C sepa de qué forma, con que argumentos y con qué tipo de retorno se va a llamar a una función, es necesario especificar su prototipo, por eso se escribió antes de main la línea:

```
unsigned int pid();
```

Esto le indica al compilador que no tiene que chequear, ni mandar argumentos a pid para invocarla, y que luego, el tipo de datos que tendrá en eax será de tipo int sin signo. Recordar que en ASM no hay tipo de datos!

Recepción de parámetros

También es necesario poder acceder a los argumentos de las funciones, por ejemplo si se implementara la función puts en una biblioteca de funciones, sería algo equivalente a:

```
//libc.c
#define STDOUT 1

int sys_write(int fd, void *buffer, int size);

int puts(const char* str) {
      int len = strlen(str);
      return sys_write(STDOUT, (void *) str, len);
}
```

Y la implementación de la función sys_write podría ser algo como:

```
; libasm.asm
GLOBAL sys_write

ALIGN 4
sys_write:
      push ebp
      mov ebp, esp

      push ebx ;preservar ebx

      mov eax, 0x4
```

```
      mov ebx, [ebp+8]  ;fd
      mov ecx, [ebp+12] ;buffer
      mov edx, [ebp+16] ;length
      int 0x80

      pop ebx

      mov esp, ebp
      pop ebp
      ret
```

Ejercicio 3

Realice una implementación para cumplir con el ejemplo (implementar nuestra propia función puts). Utilice la directiva **-fno-builtin** para indicarle a C que no debe linkeditar contra la biblioteca estándar.

Variables

Hasta ahora, siempre que hizo falta reservamos espacio en la zona de datos de un programa. Por ejemplo, para guardar una variable. Pero, ¿Qué ocurriría en el caso de que una función sea recursiva?

¿Cómo hace C a que las variables automáticas sólo vivan en el scope de la función? La respuesta es guardar esas variables en el stack, así, cuando un stack frame se destruye, al mismo tiempo se libera la memoria utilizada para sus variables.

Ejemplo 4

```
//suma.c

int suma(int a, int b) {
      int resultado;
      resultado = a + b;
      return resultado;
}
```

Una posible salida en Assembler podría ser:

```
suma:
      push ebp
      mov ebp, esp

      sub esp, 16       ;reservar espacio para las variables
      mov eax, DWORD PTR [ebp+12]
      mov edx, DWORD PTR [ebp+8]
      add eax, edx
      mov [ebp-4], eax  ; resultado
      mov eax, [ebp-4]
```

```
      leave             ;otra forma de desarmar el stack frame
      ret
```

De esta forma, la variable resultado, existe únicamente para la función, no tiene visibilidad para una función externa.

Reponder

- ¿Porqué no se debe devolver un arreglo local a una función cómo valor de retorno?
- ¿Porque restan 16 bytes de ESP y no simplemente 4?
- ¿Porqué luego, de haber obtenido el resultado de la suma, que queda en el registro EAX, se lo guarda en la zona de memoria asignada a la variable resultado y luego se la vuelve a leer de ese lugar?

Ejemplo 5

Genere la salida en Assembler con GCC de la siguiente función:

```
//factorial.c
int factorial(int n) {
      if (n == 0)
            return 1;
      int factorial_n_1 = factorial(n-1);
      return n*factorial_n_1;
}
```

Identifique a que dirección de memoria relativa corresponde la variable factorial_n_1.

Ahora es obvio que se puede llamar de manera recursiva, ya que cada invocación de la función tendrá su propia versión de dicha variable.

Ejercicios

Ejercicio 5

Escriba una función en Assembler que retorne siempre el valor 7. Llámela con C y muestre por salida estándar dicho valor.

Ejercicio 6

Enumere las diferencias de declarar las variables y utilizarlas de las siguientes formas:

1.

```
//variables1.c
int foo() {
      int numero;
}
```

2.

```
//variables2.c
int foo() {
      int numero = 21;
}
```

3.

```
//variables3.c
int numero;
int foo() {
      numero = 21;
}
```

4.

```
int foo() {
      static int numero = 21;
}
```

5.

```
extern int numero;
int foo() {
      numero = 10;
}
```

```
int numero = 21;
int bar() {
      numero = 30;
}
```

6.

```
static int numero = 10;
int foo() {
numero = 20;
}
```

Ejercicio 7

Declare en una función de C, un arreglo de las siguientes formas:

- Sin inicializar
- Con inicialización (int numeros[20] = {0}, ó char msg[] = “mensaje”)
- Sin inicializar y luego realizando una escritura en el índice 10
- Con inicialización y luego realizando una escritura en el índice 10
- De manera global, sin inicializar
- De manera global inicializando.

Primero suponga cómo debería hacer C para su declaración, y luego compare sus suposiciones con la salida en assembler.

Ejercicio 8

Suponga que ud. está implementando la biblioteca estándar de C. Basándose en la guía anterior, prepare el contexto para llamar a la función inicial (main), y luego termine el programa con el valor de retorno que haya indicado la función main. Su loader debe utilizar la etiqueta _start

Ejercicio 9

Cómo parte de la implementación de su biblioteca estándar de C, implemente las funciones que le permitan acceder al sistema operativo para:

- Cerrar un programa
- Leer de un file descriptor
- Escribir a un file descriptor (ya implementado)
- Abrir un archivo y cerrarlo

Lista de system calls:

- https://web.archive.org/web/20160213015253/http://docs.cs.up.ac.za/programming/asm/derick_tut/syscalls.html
- https://web.archive.org/web/20180806043517/http://syscalls.kernelgrok.com/

Escriba un programa que haga uso de su nueva biblioteca para abra un archivo, imprima línea por línea con el número de línea a la derecha y luego lo cierre. Recuerde compilar con la directiva **-fno-builtin**

Ejercicio 10

Escriba un programa que imprima por salida estándar el fabricante del procesador. Investigue la función cpuid de assembler.

Ejercicio 11

- Implemente la función en C fibonacci(n), que reciba un número, y devuelva el número de fibonacci que corresponda.
- Compile a mano, una posible salida en assembler de dicho programa.
- Compruebe su resultado con el de haber generado el código con **gcc**
- Haga un seguimiento, paso a paso, de la pila por cada instrucción de assembler ejecutada para un fibonacci de 3.

Ejercicio 12

Dado el siguiente programa:

```
;ejemplo.s
```

```
main:
      push  ebp
      mov   ebp, esp
      and   esp, -16
      sub   esp, 32
      mov   DWORD PTR [esp+19], 1819043176
      mov   DWORD PTR [esp+23], 1870078063
      mov   DWORD PTR [esp+27], 174353522
      mov   BYTE PTR [esp+31], 0
      lea   eax, [esp+19]
      mov   DWORD PTR [esp], eax
      call  magia
      lea   eax, [esp+19]
      mov   DWORD PTR [esp], eax
      call  printf
      mov   eax, 0
      leave

magia:
      push  ebp
      mov   ebp, esp

      sub   esp, 16
      jmp   .L4
.L6:
      mov   eax, DWORD PTR [ebp+8]
      movzx eax, BYTE PTR [eax]
      cmp   al, 96
      jle   .L5
      mov   eax, DWORD PTR [ebp+8]
      movzx eax, BYTE PTR [eax]
      cmp   al, 122
      jg    .L5
      mov   eax, DWORD PTR [ebp+8]
      movzx eax, BYTE PTR [eax]
      mov   BYTE PTR [ebp-1], al
      movzx eax, BYTE PTR [ebp-1]
      sub   eax, 32
      mov   BYTE PTR [ebp-1], al
      3mov  eax, DWORD PTR [ebp+8]
      movzx edx, BYTE PTR [ebp-1]
      mov   BYTE PTR [eax], dl
.L5:
      add   DWORD PTR [ebp+8], 1
.L4:
      mov   eax, DWORD PTR [ebp+8]
      movzx eax, BYTE PTR [eax]
```

```
    test  al, al
    jne   .L6
    leave
    ret
```

Indique lo que está haciendo, haga un seguimiento de la pila.

Ejercicio 13

Deduzca cómo se pasan las estructuras entre funciones, ¿Qué diferencia hay entre pasar una estructura por referencia o por copia? ¿Cómo es el retorno?

x86_64 ABI

Hasta ahora trabajamos con la ABI de 32 bits de C y Assembler. Básicamente los conceptos son los mismos, lo único que cambia es el pasaje de parámetros.

El pasaje de parámetros de realiza de la siguiente forma:

- Se cargan los argumentos en los registros
- Se llama a la función
- Los argumentos se copian al stack y se referencian desde ahí.

Clasificación de argumentos

Los argumentos se clasifican de la siguiente forma:

- INTEGER: char, short, int, long, long long y punteros
- SSE: floats y doubles
- MEMORY: datos mayores a un quadword (8 bytes) y datos desalineados

Pasaje de argumentos

Se pasan de la siguiente forma:

- Si el dato es INTEGER, se van ocupando los registros rdi, rsi, rdx, rcx, r8 y r9 en orden
- Si el dato es SSE, se van ocupando los registros xmm0 a xmm7 en orden.
- Si el dato es MEMORY, se pasan por stack y devuelven por stack, de izquierda a derecha (igual que en 32 bits).

Retorno de valores

Para devolver los valores:

- Si el dato es INTEGER, se utiliza rax y rdx
- Si el dato es SSE, se retorna por xmm0 y xmm1

Registros a preservar

- rbp
- rsp
- rbx
- r12
- r13
- r15

Esto implica que hay funciones que aunque tengan distinto prototipo, tienen la misma forma de pasar parámetros, por ejemplo las funciones

```
int foo(int arg1, double arg2);
```

Tiene el mismo prototipo que la función:

```
int bar(double arg2, int arg1);
```

En síntesis:

```
typedef struct {
      int a, b;
      double d;
} structparam;

structparam s;
int e, f, g, h, i, j, k;
long double ld;
double m, n;
__m256 y;

extern void func(int e, int f,
                 structparam s,
                 int g, int h,
                 long double ld,
                 double m,
                 __m256 y,
                 double n,
                 int i, int j, int k);

func(e, f, s, g, h, ld, m, y, n, i, j, k);
```

Los registros van a quedar de la siguiente forma:

General Purpose Registers	Floating Point Registers	Stack Frame Offset
%rdi: e	%xmm0: s.d	0: ld
%rsi: f	%xmm1: m	16: j

%rdx: s.a, s.b	%xmm2: y	24: k
%rcx: g	%xmm3: n	
%r8: h		
%r9: i		

Ejercicios

Consigna

Realice los ejercicios de pasaje de parámetros, pero esta vez en 64 bits.

Para compilar, reemplazar los argumentos **elf32** por **elf64** de **nasm** y **-m32** por **-m64** en gcc.

Ejercicios de debugging

Tome los archivos **ej_debug_1.asm** y **ej_debug_2.asm** disponibles en el campus, estos son programas assembler que usan funciones de librería estándar de C. Compile estos programas en 32 bits.

1. El programa ej_debug_1 debiera imprimir todos los argumentos que recibe. Ejecute el programa ej_debug_1 con y sin argumentos. Verá que cuando recibe argumentos el programa cancela. Diagnostique el problema y solucione el código.
2. El programa ej_debug_2 debiera imprimir el número 1234567890 pero también falla. Diagnostique el problema y solucione el código.

Ejercicio 14

Dado el siguiente código en C

```
void encrypt(char *plain, char *cipher){
      if(!(*plain)){
            *cipher = 0;
            return;
      }

      char pad = get_pad();
      *cipher = *plain + pad;
      encrypt(plain+1, cipher+1);
}

void test(){
      char *msg = “Ark”;
      char cipher[] = “--------”;
      cipher[4] = ‘0’;
```

```
      encrypt(msg, cipher);

      printf(cipher);
}
```

- Suponga que el dispositivo de encriptación comienza devolviendo 1 byte igual a 0 e incrementa en 1 en cada llamada.
- Muestre como cada operación altera el estado de la pila al invocar la función test y cuál es la salida por salida estándar.
- ¿Encuentra algún error posible en todo el código anterior?

Ejercicio 15

Sea la siguiente función escrita en C para un procesador con arquitectura x86:

```
int factorial(int n) {
      int aux, fact;

      if(n==0)
            return 1;
      aux = n;
      fact = factorial(n-1);
      return fact * aux;
}
```

- Mostrar el estado de la pila si es invocado con n = 3.

Ejercicio 16

Considere una función en ASM de 32 bits denominada CheckLong. A esta función se le pasan dos parámetros por la pila:

- El primer parámetro es la dirección de inicio de un vector de caracteres.
- El segundo parámetro es un valor entero que indica la cantidad de elementos del vector de caracteres.

La función CheckLong debe recorrer el vector hasta encontrar un cero, luego compara con el valor siguiente al cero (el cual debería ser la cantidad de elementos del vector) y verifica si la cuenta es correcta comprando este número con la cuenta realizada.

CheckLong retorna la diferencia que hubo entre el valor calculado y el valor informado. Es decir, si son iguales retorna cero, si el valor calculado es mayor al informado retorna la deferencia en valor positivo, si el valor informado es mayor, retorna la diferencia en valor negativo.

Además se le pide a usted que esta rutina tiene que quedar lista para poder ser llamada desde un programa en C.

Se pide:

1. Indique de qué manera recibirá CheckLong cada uno de los parámetros a la función.

2. Programar utilizando lenguaje ASM el código de la función CheckLong usando el siguiente fragmento de programa:

```
section .data
msg:  db “Hola Mundo”, 0
len:  db 10
```

3. Escriba un programa en C en un archivo llamado principal.c que sólo llame a la función realizada en el punto #2 y que imprima en pantalla si el chequeo de la longitud fue exitoso o no.

4. ¿Qué consideraciones debe tener para poder compilar y linkeditar ambos archivos, el archivo **.asm** y el archivo **.c**?

Ejercicio 17

Dado el siguiente programa que se compila para un procesador de 32 bits:

```
#include<stdio.h>

int calculo(int param1, int param2, char tipo){
      int resul;

      if(tipo==’s’)
            resul = param1 + param2;
      else
            resul = param1 * param2;

      return resul;
}

int main(void){
      int valor1 = 7;
      int valor2 = 3;
      char operación;

      printf(“Ingrese el tipo de operación: ‘s’ suma y ‘m’
multiplicar:\n”);
      scanf(“%c”, &operacion);
      printf(“Resultado: %d\n”, calculo(valor1, valor2, operacion));
      return 0;
}
```

1. Explique con dibujos respectivos el contenido de la pila durante toda la ejecución del programa.

2. Muestre los registros del microprocesador que intervienen en los dibujos de la pila.

Ejercicio 18

Se le pide que escriba una función en assembler de Intel. Usted debe realizar una rutina que será llamada desde C y tiene el siguiente formato:

```
int imprime_pantalla(char *encabezado, int tam_enc, char *pie, int tam_pie)
```

Esta rutina debe imprimir dos cadenas de caracteres en una pantalla de 480 caracteres monocromáticos (80 columnas y 6 filas) que se encuentra mapeada a partir de la dirección de memoria A0000000h. Cada posición de memoria representa un carácter de la pantalla.

La rutina imprime una cadena llamada encabezado en la primera fila de la pantalla y otra cadena llamada pie en la última fila de la pantalla. Además recibe el tamaño de cada cadena a imprimir, el cual es un número que puede ir de 1 a 80 como máximo. Si la rutina recibe un valor fuera de ese rango retorna el valor 1. Si la rutina se ejecuta sin problemas retorna el valor cero.

Como caso de uso y para contestar lo pedido en el ejercicio usted probará con la cadena encabezado “Este es el título” con 17 como tamaño de cadena, y la cadena pie “Fin de mensaje” con 14 de tamaño.

Se pide **en este orden** que:

1. Dibuje el estado de la pila al iniciar la rutina con cada una de las direcciones de memoria y contenidos de la pila, utilizando el caso de uso propuesto.
2. Realice la rutina en assembler, pero **solo implemente** la parte que imprime la cadena pie. **No** escriba la parte de código que imprime el encabezado.