What is the Kernel?

In this unit, we will begin an in-depth exploration of the Linux kernel. The first and most obvious question is: just what exactly is the "kernel"?!

The Linux kernel is an unusual type of program. Most programs to which we are accustomed have one point of entry, run for some time, and then terminate. The kernel has one initial point of entry which is used at boot-time, and then multiple points of controlled re-entry.

The kernel is compiled and linked just like an ordinary C program, in that it consists of many individual object (`.o`) files which are compiled from source code (the Linux kernel is written exclusively in C and assembly). Within the kernel is the equivalent of a `text` section, containing the executable code, as well as `data` section with variable initializers. Like an ordinary C program, when the kernel starts it sets aside space for uninitialized `bss` variables, and can dynamically allocate memory for other data structures. Also, like a user-level C program, the kernel can dynamically load and unload executable modules.

Unlike the C environment that is documented in the C language standards documents, the kernel does not have the standard library. There is no printf, no malloc, no fopen, etc. because these standard library functions would require an operating system to provide the services. There is a subset of library functions which is linked in the kernel such as strcpy. The kernel is also coded to avoid all use of the floating point registers, so functions such as pow() and sqrt() would never be seen in kernel code, and indeed the keywords `float` and `double` would generally be absent too.

At this point, it might be instructive to look at how the Linux kernel source code is arranged. The kernel can be compiled for a variety of target architectures. From the top-level directory, we see the following directories, each of which contains architecture-independent code. Thus the code below is only in C, not assembly. There is one top-level directory called `arch`, under which is one entry for each supported architecture, and below each of those is another set of subdirectories with similar structure to the architecture-independent set. Here is a description of these directories:

- `block`: A fairly small set of utility routines pertaining to block-level access to hard disks.
- `crypto`: Encryption functions.
- `drivers`: A vast set of routines for interfacing with I/O devices, including arch-neutral stuff such as the SCSI messaging layer, and support for specific peripherals, motherboards, etc.
- `fs`: Routines to support the file system, such as open, read, write system calls. Also subdirectories for each supported file system type module.

• `include`: Header (.h) files
• `init`: Code which is executed when the system first boots.
• `ipc`: System V and POSIX IPC mechanisms
• `kernel`: A considerable amount of code including all of the synchronization, task control, signals, halting and restarting the system, etc.
• `lib`: A number of utility routines needed by different subsystems, including synchronization primitives such as spin locks and semaphores.
• `mm`: The virtual memory subsystem.
• `net`: Networking protocols
• `security`: Some extra stuff for "security-hardened" versions of the kernel.
• `sound`: Sound drivers and support

(there are a few other subdirectories which have to do with the kernel build and install process but are beyond the scope of this course.)

Boot-up

The following describes how the linux kernel under X86-32 bit architecture is booted:
• The linux kernel is built from source and often included as a binary distribution. A valid kernel must be present to support the particular architecture of the machine that is being booted, and the particular device drivers that are needed at boot time. The kernel source code as described above is first compiled into an a.out file. However, a.out files mean nothing to the firmware, so additional preparation is necessary. The *image* of the text and data sections is pulled out of the compiled kernel. Typically, to save space on smaller bootable media, the kernel image is compressed. A bootable kernel consists of a small, uncompressed portion of bootable code followed by the compressed text/data image. This bootable kernel (typically called `vmlinuz` where the "z" stands for compressed) is installed on the bootable medium (hard disk, CD/DVD ROM, USB stick).
• To boot the system, the firmware ("BIOS") causes the kernel image to be read in from the boot device (e.g. the hard disk) into a specific place in physical memory. The exact manner in which this happens may be discussed in a later unit. The firmware then transfers control to the starting address of the kernel.
• BIOS runs in supervisor mode, with address translation turned off (so all addresses used in code are physical) and with interrupts disabled. Hand-off to the kernel image is made in this condition. Thus the first part of the kernel's initialization routines have been compiled and linked with a specific address in mind which corresponds to the load address. The next task is to de-compress the image which is in one place in physical memory and write it to another location. For the (32-bit) x86 architecture, this is physical address `0x00100000`. A jump is then made to that physical address.
• At around this time, the kernel turns on virtual address translation. On the x86 32-bit architecture, the Linux kernel uses virtual addresses below 0xC000000 exclusively for user-level processes, and reserves the 0xC000000-0xFFFFFFFF range exclusively for

kernel memory. We'll see how this is exploited to accomplish fast access to kernel data structures. **Note: All parts of the kernel see the same virtual address mappings.** To get the kernel running on virtual, rather than physical addresses, page tables are created which map 0xC00000000 to physical address 0x00100000, and so on for higher addresses, until the entire static part of the kernel (text, data, bss) has been mapped. The static bss pages are explicitly zero-filled at this time. A temporary stack is set up to allow the kernel initialization code to run.

• The MMU is turned on and thereafter the kernel runs on virtual addresses.

• The physical page frames that are not part of the static part of the kernel become the page frame pool. The kernel is able to dynamically allocate memory for itself out of this pool, in addition to satisfying user-level page fault demands.

• The interrupt vector tables are appropriately initialized (see "Interrupt Handling in Hardware" below).

• The kernel then calls initialization routines for its various subsystems, e.g. virtual memory, file systems, scheduling. They allocate dynamic kernel memory as needed. An inventory of hardware devices is taken and the initialization routines for each device are called.

• Interrupts are now enabled.

• Now the kernel is able to provide its services, but has no user-mode processes to which they can be provided yet. The device which contains the root filesystem is determined from information passed during boot loading, and the root filesystem is mounted.

• The kernel creates process #1 and causes that process to exec/map a specific binary program (`/sbin/init`), running as the super-user (uid and gid = 0). This "init" program must reside on the root filesystem, and is the bridge between kernel initialization and user-mode initialization. It remains running for the life of the system.

• pid 1 is marked as ready to run and the kernel relinquishes control to the process scheduler (unit 9 and 10) which immediately selects pid 1 to run (since there is nothing else to do!) Control now exits kernel mode and the init process begins to execute at user level. Of course control immediately re-enters the kernel as init page-faults in its first page of executable text.

• First, `init` runs a series of programs and shell scripts which complete the initialization of the system. This includes locating and mounting other filesystem volumes as needed. Then, init spawns other user-mode programs which ultimately provide access to the machine for the end-user.

When booting up a multiprocessor machine, the BIOS starts the machine in single-processor mode. Fairly late in the kernel's initialization, the other processors are enabled. They are initially given the *idle task* to run. Once additional processes other than pid 1 are runnable, the other CPUs will start to pick up their loads.

Re-entering the kernel

We have seen that there is one initial entrypoint to the kernel, through the boot process. Thereafter, the kernel relinquishes control to user-mode processes. The kernel is entered again only through the interrupt mechanism.

We can broadly divide re-entry into two categories:
• **Synchronous**: When the interupt is raised because of the execution of a specific instruction, that is said to be synchronous. The Linux kernel calls these synchronous events **Exceptions**. Other similar terms are "Fault" and "Trap". Exceptions are said to occur in the context of a particular running process. The kernel associates a unique kernel-mode stack for each process (or to be precise, for each user-level thread), and so we can think of the kernel as containing one kernel-mode thread of control for each user-level thread, which gets activated when an exception is raised. Thus the handling of the exception in the kernel is in effect a controlled extension of the user-level thread into the kernel.
• **Asynchronous**: These events are not correlated with a specific instruction, but come from hardware devices. The Linux kernel calls these "Interrupts", which is somewhat ambiguous because Exceptions are also a form of interrupt. Asynchronous interrupts are handled on whatever kernel-mode stack happens to be active at the time.

In most operating systems literature, synchronous entries to the kernel are said to be "*top half*," and asynchronous are said to be in the "*bottom half*". Unfortunately the Linux kernel uses the term "bottom half" in an inconsistent manner, so we will avoid top/bottom half terminology to reduce confusion.

Exception Types

• Many exceptions are the result of program error. These are also called **faults**. A fault is defined as a condition which prevents the current opcode from being executed. The default behavior of the kernel for most faults is to post a signal to the offending process. Examples of such faults include: integer divide by 0 and illegal instruction.
• A **Page Fault** exception occurs when the hardware was unable to perform a memory access. On the x86-32 architecture, this same fault type is used both for when a valid PTE can not be found and for a protection violation, the distinction between the two being passed to the fault handler by hardware in a specific register. Page Faults are quite normal and not necessarily a sign of program error, as discussed in Unit 5.
• The kernel uses some exceptions to be clever and efficient about things. For example, the kernel usually does not bother with floating point registers. It turns the floating point unit off, and then if the process tries to use a floating-point operation, it raises an exception and the kernel knows that it needs to worry about it for that process. So these exceptions are like a Page Fault, in that the kernel’s handler must determine whether the exception is benign or the result of an errant program.

• When faults are resolved successfully by the kernel, control returns to user level and the previously faulted instruction is re-tried, and this time should succeed.
• A **System Call** is a particularly important type of exception, and we'll spend some time looking at system calls in detail.

Interrupt Types

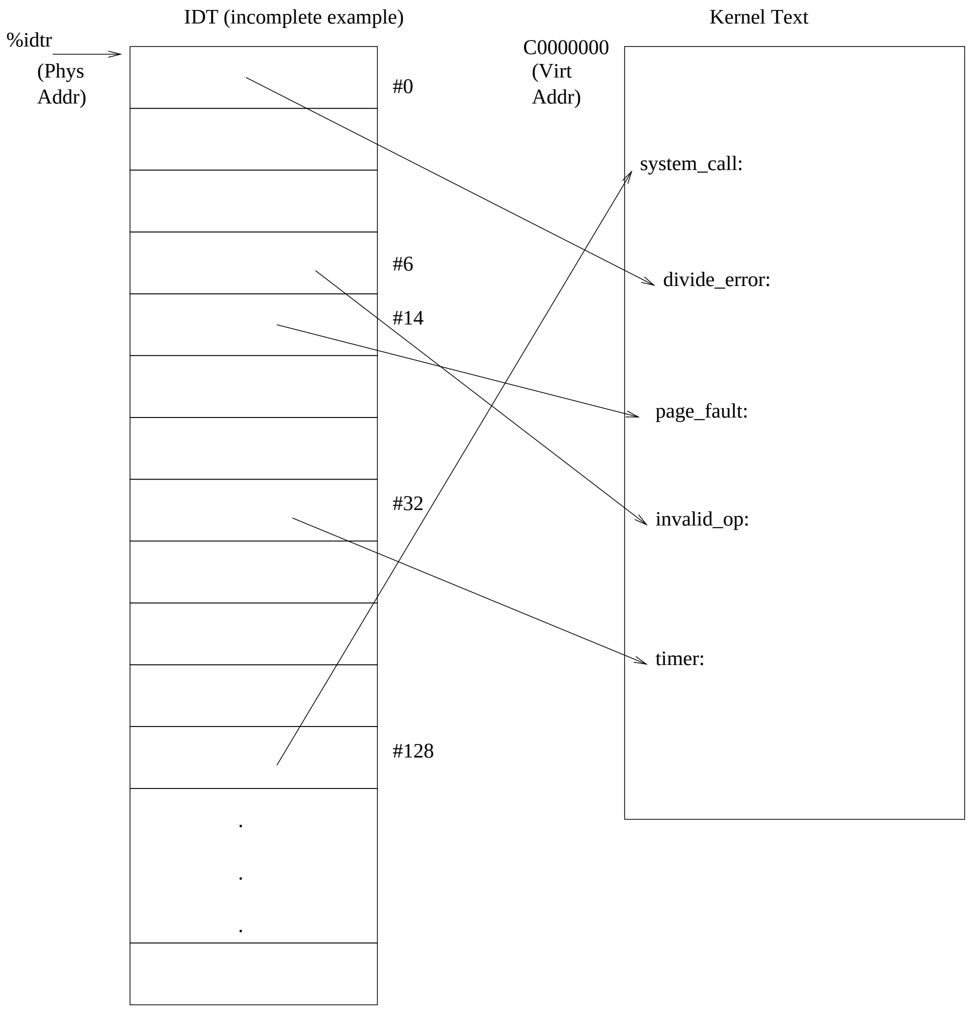
• Hardware devices will raise interrupts to indicate that they have entered a ready or non-ready state, that data are available, or that a data transfer operation has completed.
• The kernel programs a chip on the motherboard to deliver a periodic "heartbeat" interrupt, e.g. every millisecond. This **Periodic Interval Timer (PIT)** is very important to the scheduler. Other time-based events in the kernel (e.g. network protocol retransmission timers) are derived from this clock source.
• On a multi-processor system, one processor may execute a series of instructions which cause a hardware interrupt to be posted to the other processors. This is known as an **Inter-processor Interrupt (IPI)**. The Linux kernel uses this in several specific cases: (a) To shut down the system (b) to request that the other processors re-examine their task scheduling (covered in next unit) (c) to purge stale TLB entries.
• Hardware will deliver interrupts to notify the kernel of important system events, such as a hardware failure, battery power failure, etc. If continued system operation is possible, the kernel will ultimately translate these into user-mode events. Otherwise the kernel will print out an informative message and halt the system.

Important Shared Processor/Kernel Data Structures

The behavior of the CPU is hard-wired (or programmed in its microcode) and thus immutable to any software code including the kernel. The kernel is compiled for a specific architecture and knows about data structures that the processor expects to see in memory, which control how the processor reacts to certain events. Because the processor itself must respond to virtual memory lookup failures (page faults) these data structures are typically defined at physical, rather than virtual memory addresses. On the X86-32 bit architecture, these data structures are:
• The **Interrupt Descriptor Table (IDT)** is an array of 256 interrupt descriptors. The physical address of this array is set via the special `idtr` register, which can only be accessed through special privileged instructions LIDT and SIDT. Although the structure of the IDT is baroque because of legacy X86 segmented memory model issues, it is basically a table of program counter (%eip) addresses, which are the handler addresses (virtual addresses) for each of the 256 possible interrupt or exception/fault vectors. The kernel has initialized all IDT entries to point to valid kernel functions before allowing user-level code to begin execution after system boot. The IDT is generally not modified after this.
• The **Task State Segment (TSS)** is a small data structure which resides inside a larger

data structure known as the Global Descriptor Table (GDT). The Linux kernel does not utilize all of the functionality that is available with the TSS, but uses it simply to control the stack pointer address that will be used during interrupt/exception handling. The special `tr` register controls the location of the TSS and is generally set once. On multi-processor systems, there is a TSS for each processor.

• The page table, which has been described conceptually in previous units, and will be described in greater detail in a subsequent unit. The `cr3` register controls the physical address of the highest-level page table (which linux calls the Page Global Directory).



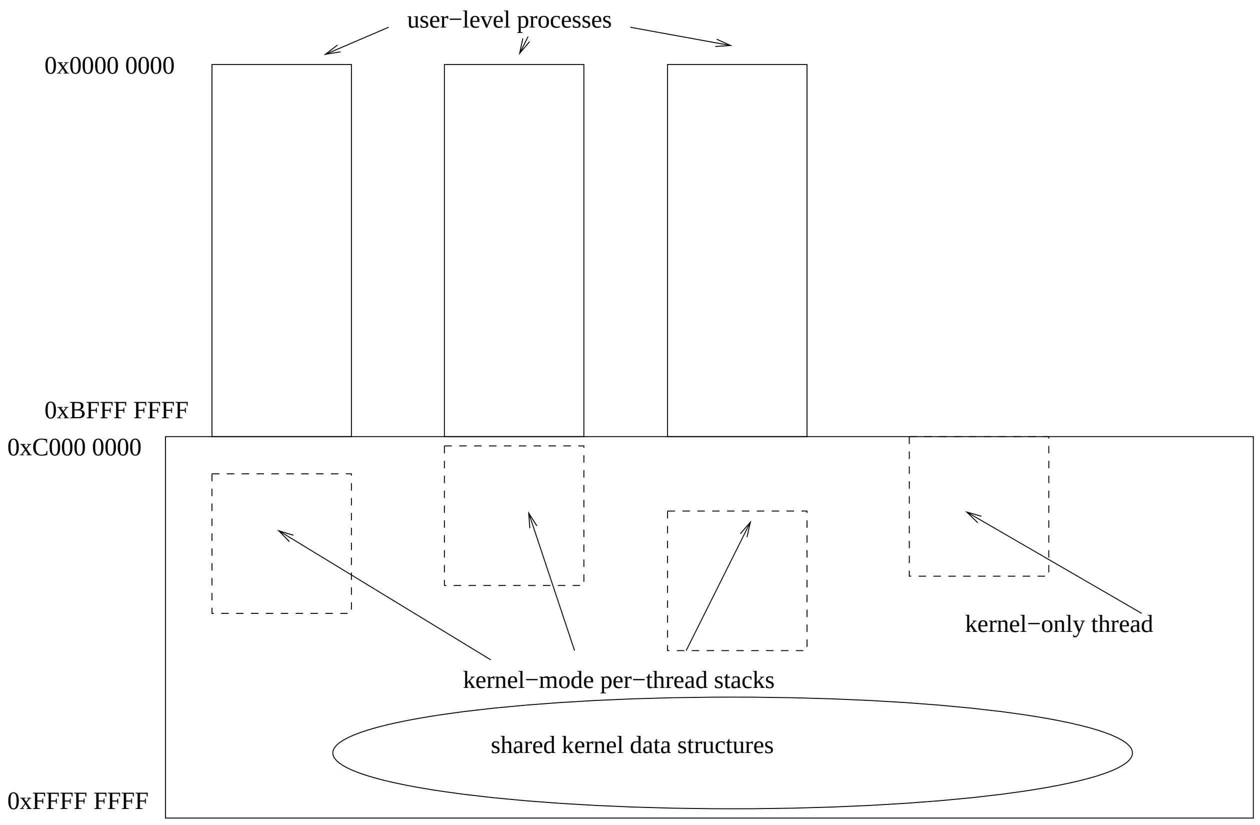
Interrupt and Exception handling in hardware

On the 32-bit X86 architecture, the following steps are taken by the processor in response to an interrupt or exception:

• Each interrupt or exception has an associated **vector** between 0 and 255. This is used to index the IDT, from which the handler address is fetched. The handler address is the

virtual address of the first opcode of the associated handler function within the kernel. This function is generally written in assembly language. The kernel has made sure that all 256 entries have valid handler addresses. In the X86 architecture, vectors 0-31 are reserved for processor-generated exceptions, while 32-255 are user-defined and typically used for I/O devices (the system call vector, 128 decimal, falls within this range)
• When the processor is currently running in User mode, it fetches the TSS. Within the TSS is the new value (virtual address) of the stack pointer. This will be the kernel-mode stack, as described later. The kernel maintains a separate kernel stack for each process, and makes sure the correct stack address was placed within the TSS before performing a context switch. If however, the processor was already in Supervisor mode at the time of the exception/interrupt, then the stack pointer is already within a valid kernel stack, and the TSS is not used by the processor.
• The processor now transitions to Supervisor mode, and the %esp register is pointing within the kernel-mode stack for the task.
• The processor pushes onto the kernel-mode stack the value of the old stack pointer register `%esp`, the flags/status register `%eflags`, and the program counter register `%eip`. The `%cs` and `%ss` registers (which have to do with how code and stack memory are accessed) are also saved. This set of 5 registers is critical because it is what the hardware relies on to determine where the stack is, and where the next instruction is.
• For certain types of exceptions, the processor pushes an error code onto the stack. This is used, e.g., to distinguish between a page translation fault and a page protection fault.
• The program counter location of the handler, which was fetched from the IDT, is now loaded into the `eip` register, effecting a jump to that entrypoint.
• Recall that the kernel controls the page tables, and establishes them for each process. The kernel virtual address space, 0xC0000000-0xFFFFFFFF, is always present in these page tables, but the Privileged flags in the Page Table Entries are set so user processes can not directly access the kernel memory (this would be really bad....however user processes running as root can use a pseudo-device `/dev/kmem` to access kernel memory as if it were a file). This page table arrangement means that as soon as control enters the kernel, the shared kernel address space is immediately available.

Any interrupt or exception handler ultimately terminates by executing the special `iret` instruction, which:
• Pops the 5 registers from the stack which had been saved by hardware.
• Resets the privilege level to the previous value (because the eflags register has been restored from the stack)
• Resumes execution with the stack pointer now set back to the original user-mode stack. Interrupts occur between instructions, and so execution resumes at the next instruction. For exception returns, if the instruction caused a fault (e.g. Page Fault) then that instruction is re-started, otherwise the next instruction is executed.



Kernel control flow paths

Now that we've seen the types of kernel entrypoints, lets continue further to talk about how control flows within the kernel and back out again.

At any given moment, a CPU is either:
• Executing a user-level thread (process)
• Executing a kernel thread. This is identical to a user-level thread except that the virtual address space is entirely within the kernel.
• Handling an exception
• Handling an interrupt
• Temporarily halted because there is nothing else to do at the moment. We call this the "Idle" state.

A **context switch** means changing from one thread (either kernel or user level) to another. *Context switches can only occur at the instant of returning from an interrupt or execption*.

While handling an exception or interrupt, it is possible that another exception or interrupt will arise. Thus the flow of control in the kernel is multiply re-entrant, and nested. E.g.

while handling a system call (exception), a keyboard interrupt is received. The exception handler is suspened (by hardware) and the keyboard interrupt handler begins to run. While servicing that, a disk interrupt is received and handled. When the disk handler finishes, the keyboard handler resumes and finishes, then the system call is allowed to continue.

To simplify kernel programming and synchronization issues, the Linux kernel is carefully coded so that kernel code never produces exceptions, with one caveat: During the handling of a system call (exception), a Page Fault (exception) may be raised. Thus we can say that exceptions will never arrive during interrupt handling, and will never arrive during exception handling other than a Page Fault during a System Call. An interrupt may occur at any time, because it is asynchronous.

Making a System Call

We are now ready to discuss one particular kernel control path: the system call. We'll illustrate a fairly simple one: `getuid()`. A user-level program calls `getuid()` as an ordinary C function. This function is provided by the standard C library which is linked with all C programs. The `getuid()` function is written partially in C and partially in assembly language. This provides the "glue" between the user-level domain of the C program, and the kernel's system call API.

The 32-bit X86 API

Argument passing between user-level functions in C (X86-32) is via the stack. However, during a system call, the processor will be switching to a different, kernel stack. The user-level program obviously can not write to the kernel stack. Conversely, although the kernel *can* access the user-level stack memory, it would rather not, since that might create a Page Fault. The solution is to pass arguments to system calls in registers. The convention used on the x86-32 architecture is that the 32-bit arguments are passed such that the first argument is in the `ebx` register. The second is placed in the `ecx` register. The third through sixth are placed in registers `edx, esi, edi, ebp` respectively. If the number of arguments to the system call exceeds 6 (rare), then all of the arguments are placed on the user-level stack and the kernel receives just the address of that argument block.

Each available system call is assigned a specific number by the kernel. A given system call number also has a specification for what arguments are expected. The standard C library must therefore be compiled, at least in part, with an eye towards the exact architecture on which the program will be run. As the kernel evolves towards higher version numbers and new system calls are added, backwards-compatability with older code must be maintained. System call numbers are not re-used. If it is necessary to change the semantics of a system call, a new call is defined with a new number, and the kernel provides both versions. In the Linux 2.6.15 version kernel, there were 294 defined system call numbers, by version 2.6.23 that number had grown to 325, and by 2.6.32 it had reached 338!

The system call number is passed to the kernel in the `eax` register. Now the user-level `getuid` function is ready to actually make the system call. There are no arguments to this particular system call, so the system call # corresponding to getuid (199 decimal) is put in eax and then a special instruction is used. There are two ways to make a system call: using the `INT $0x80` software interrupt exception instruction, or using the `SYSENTER` instruction. We'll follow the former example. The reader is referred to the book *Understanding the Linux Kernel* for more information on the SYSENTER method.

The `INT $0x80` instruction causes an exception to be raised with vector code 128. The hardware then vectors to the kernel entrypoint in the IDT for vector 128, which the kernel had previously initialized to point to the (symbolic) kernel virtual address `system_call`. The code below is a simplified version of /usr/src/linux/arch/x86/kernel/entry.S:

```
system_call:
         pushl    %eax                       #contains system call #
         SAVE_ALL                            #macro to push important
                                             #registers on the stack
         movl     $0xFFFFE000,%ebp           #mask SP to get to
         andl     %esp,%ebp                  #thread_info
         testw    $_TIF_SYSCALL_TRACE,TI_FLAGS(%ebp)    #test thread_info.flags
         jnz      syscall_trace_entry        #for syscall tracing on
         cmpl     $nr_syscalls, %eax         #bounds check
         jae      syscall_badsys
         call     *sys_call_table(0,%eax,4)  #indirect addressing
         movl     %eax,PT_EAX(%esp)          #poke return code into EAX slot
syscall_exit:
         cli                                 #temporarily mask interrupts
         movl     TI_flags(%ebp),%ecx        #get flags field of thread_info
 #ALLWORK_MASK includes all TIF_XXX thread info flags that indicate more
 #work might be needed before returning to user space
         testw    $_TIF_ALLWORK_MASK,%cx     #see if any flags are set
         jne      syscall_exit_work          #if so more work before exit
restore_all:
         RESTORE_REGS                        #pop registers from stack
         addl     $4,%esp                    #discard original eax
         iret                                #return from interrupt
syscall_badsys:
         movl     $-ENOSYS,PT_EAX(%esp)      #poke error return code
         jmp      syscall_exit               #simplified
syscall_exit_work:                           #simplified
         testb   $_TIF_NEED_RESCHED,%cl      #flags already in ecx
         jz       work_notifysig             #if clear, must be signal pending
work_resched:
         call     schedule                   #otherwise, task switch
         cli                                 #avoid missing an interrupt
         movl     TI_flags(%ebp),%ecx        #check flags again
         andl     $_TIF_WORK_MASK,%ecx
         jz       restore_all                #OK to return to userland
         testb    $_TIF_NEED_RESCHED,%cl
         jnz      work_resched               #still need rescheduling
work_notifysig:
         #we’ll see this code in a later unit
```

The `SAVE_ALL` macros pushes all of the registers that the kernel is likely to clobber. In conjunction with the hardware pushes and the instruction `pushl %eax` just above, the kernel stack now looks like this:

```
0x00(%esp) - ebx            general-purpose registers,
0x04(%esp) - ecx            saved by
0x08(%esp) - edx            SAVE_ALL
0x0C(%esp) - esi            "
0x10(%esp) - edi            "
0x14(%esp) - ebp            "
0x18(%esp) - eax            " (will be syscall return value)
0x1C(%esp) - ds                     "
0x20(%esp) - es                     "
0x24(%esp) - fs                     "
0x28(%esp) - orig_eax       orig system_call number (used for syscall restart)
0x2C(%esp) - eip            program counter in user land, saved by hw
0x30(%esp) - cs             code segment register, saved by hw
0x34(%esp) - eflags         flags register, saved by hw
0x38(%esp) - oldesp         user-land stack pointer, saved by hw
0x3C(%esp) - oldss          user-land stack segment reg, saved by hw
```

The next instruction places a mask into register `ebp` and applies that mask to the stack pointer `esp`. Normally, in C programs, the ebp register has a very important function as the local frame pointer. However, we are still in an assembly language entrypoint, and thus ebp is available as a scratch register. The purpose of this masking operation deserves a considerable detour:

Kernel-mode stack and the thread_info structure

Recall that the kernel allocates an individual stack area for each user-level thread of control. Depending on kernel version and compilation parameters, the size of this stack may be 4K or 8K. For simplicity, we will assume an 8K stack. While this seems rather small, bear in mind that every bit of code in the kernel is carefully controlled. Large amounts of local variable space are discouraged, recursive programming is never used, and the depth of function call nesting rarely gets obnoxious. Therefore, the kernel programmers can rely on the fact that this 8K stack will not overflow.

Now, to compound this trickery, the kernel sticks a small data structure called `struct thread_info` at the limit of the stack, i.e. at the lowest memory address. The kernel stack pointer value stored in the TSS memory area by the kernel for each process/thread is the highest address of the allocated stack area. On entry to the kernel through an exception or interrupt, the kernel stack is empty, and the kernel stack pointer is thus furthest away from this `thread_info` data structure, and as kernel functions are called, the stack pointer gets closer to it, but should never be in any danger of over-writing it.

This arrangement of memory addresses means that on entry to the kernel, a simple masking operation of the stack pointer `%esp` yields the beginning of the `thread__info` structure. That address is kept in the `%ebp` register for a while. Also, at any point in the kernel, the inline function `current_thread_info()` performs that same masking operation. Let's look into the `thread_info` struct (/usr/src/linux/arch/x86/include/asm/thread_info.h):

```
struct thread_info {
        struct task_struct      *task;          /* main task structure */
        struct exec_domain      *exec_domain;   /* execution domain */
        unsigned long           flags;          /* low level flags */
        unsigned long           status;         /* thread-synchronous flags */
        __u32                   cpu;            /* current CPU */
        int                     preempt_count;  /* 0 => preemptable */
        mm_segment_t            addr_limit;     /* highest valid VA */
          void                        *sysenter_return;  /* don't worry */
        struct restart_block    restart_block;  /* for syscall restart */
 #ifdef CONFIG_X86_32        /* Don't worry about these next two obscure lines */
        unsigned long           previous_esp;
        __u8                    supervisor_stack[0];
 #endif
          int                         uaccess_err;       /* err from accessing user mem*/
};
```

There isn't much room on the kernel stack, so `thread_info` is pretty small. What is kept in there is only what is needed by the assembly language entry/exit routines. The kernel needs to keep a lot more information about a process, so it allocates another data structure called `struct task_struct` which is pointed to by `thread_info`. This is an unfortunate and confusing choice of names!

On slightly older Linux kernels, the `task_struct` for the process in whose context the kernel is running was accessed directly by a series of macros which computed the `thread_info` address from the stack pointer, then fetched the first word of that structure to obtain the pointer to the current `task_struct`. On later kernels, each CPU maintains a small per-CPU private memory area, and within that area maintains a global variable called `current_task`.

It's the responsibility of the part of the kernel called the *scheduler* to maintain the `current_task` variable and the corresponding structure for each task (aka process, aka thread) on the system. We'll look inside this structure in the next unit.

The X86-64 API

Although the examples herein are for the older 32-bit X86 API, also known as i386, we should examine how things change when using the 64-bit API, known as X86-64.

At user level, the first 6 arguments to a function are passed in registers, not the stack. The argument slots are registers %rdi,%rsi,%rdx,%rcx,%r8,%r9. The X86-64 API does not use the INT 0x80 instruction, but a new instruction called `SYSCALL`, which is somewhat faster. It performs the following steps in hardware:

• Save the return address (%rip register) in register %rcx (overwriting its value)

• Save the current value of the flags register (%rflags) in %r11 (overwriting it too)
• Make some adjustments to the %cs and %ss registers to allow kernel code to execute properly.
• Set the processor to privileged mode
• Load the value of a special register (MSR_CSTAR) into %rip. This privileged register has been pre-loaded by the kernel to point to the system call entrypoint.

Note that in this 64-bit API, the hardware does not stack any registers when performing a system call.

Because the %rcx register is clobbered by the SYSCALL instruction, the kernel's system call convention specifies that the arguments to the system call are passed in registers: %rdi,%rsi,%rdx,%r10,%r8,%r9. The system call number is passed in %rax. Therefore, the user-level "glue" code takes the 4th argument in %rcx and moves it into %r10, and adds the system call number in %rax. The kernel's system call assembly-language entry code will put the 4th argument back into %rcx from %r10 prior to dispatching to kernel C functions via the system call table.

Unlike the 32-bit API, the 64-bit SYSCALL API does not change the stack pointer. Upon entry via any kernel entrypoint, the kernel uses the `SWAPGS` instruction to interchange the user-mode value of the %gs register with a "hidden" %gs register that the kernel has pre-configured. This register can be used in conjunction with an obscure aspect of X86 programming known as "segment override", allowing access to a reserved area of physical memory that the kernel has pre-configured. It contains an array of scratchpad areas, one for each processor, known as the "per_cpu" areas. The value of the kernel stack pointer was saved in this per-cpu area before control was returned to user mode. Therefore, on entry to the kernel, this saved kernel stack pointer is retrieved and processing of the system call (or fault or interrupt) continues within the kernel in a normal C environment.

The kernel of course has saved all registers once it has established the kernel stack. Prior to return to user mode, the kernel makes sure that the kernel stack pointer is stored in the per-cpu scratchpad area, executes the SWAPGS instruction again to save the hidden gs register, and restores the user-mode %rip and %rflags values (which were saved on the kernel stack) into %rcx and %r11 respectively. The `SYSRET` instruction is then used to reverse the effects of `SYSCALL` and return control to user mode.

If the above description of X86-64 caused pain and/or confusion, do not panic. We will conduct the rest of our examples in 32 bits.

Now, back to our system call, already in progress

Having computed the address of the thread_info structure, the next line of assembly examines a bitwise flags word. There are quite a few TIF_XXX flags defined. Of interest here is a tracing hook: if the TIF_SYSCALL_TRACE flag is set, then the thread making the system call is being traced (e.g. through the strace command) and the kernel diverts to an alternate entry which will record the parameters of the system call and pass them back as an event to the tracer. We won't go down that road, however.

The next two lines are very important, as they illustrate data validation. Recall that the user-level process is completely untrusted as far as the kernel is concerned. If the system call number passed by the user is greater than the highest system call number (or negative...think about two's complement) then the syscall_badsys code is jumped to. This places an error code into the slot in the stack where the user's eax register was saved. Upon exit from the system call handler, this value will be popped into %eax.

System Call Return Value

On the X86 architecture, when a function returns an int (or other 32-bit value such as a pointer), that value is returned in the eax register. All return values from kernel system calls are in fact signed integers. A negative value is used to indicate an error, and that value is -error_number.

Therefore, an invalid system call will return the value -ENOSYS via the %eax register. Now, an unfortunate history lesson crops up. The UNIX API, for reasons that may be lost to time, specifies that when system calls fail, they should set the global variable errno, and return -1. So the user-level glue function does this (pseudocode):

```
int generic_system_call(arguments...)
{
          put arguments into registers
          put system call # into %eax
          INT $0x80
          if (%eax<0)
          {
                    errno= -%eax;
                    return -1;
          }
          return %eax;
}
```

Kernel system call hand-off to C

However, we know in this case that a valid system call number was used. We have been tracing out assembly language code, but most of the kernel is written in C. The instruction

```
          call    *sys_call_table(0,%eax,4)
```

uses the X86 indexed register offset indirect addressing mode as follows: The %eax register (containing the system call #) is multipled by 4 (the sizeof a pointer), and that offset is added to the base address of a table of function pointers sys_call_table.

The result of that addition is used to fetch 4 bytes from memory, and _that_ address is the one which is called as a subroutine. It is *as if*:

```
  /* Declare and initialize array of function pointers */
 /* The system call names and positions are purely an example */
  void (*sys_call_table[])()={sys_open,sys_read,sys_close,....};
   /* Hand-off to syscall handler, pseudocode */
         (*sys_call_table[%eax])(args);
```

When the kernel is compiled, the sys_call_table is filled in with the name (i.e. the virtual address) of each C function which implements each system call. It is the convention that a system call which is known as XXX to the user is implemented by a kernel function called sys_XXX.

```
asmlinkage long sys_getuid(void)
{
        return current->cred->uid;    /*cred is the credentials info */
}
```

This system call is coded purely in C, and is in fact found in the architecture-neutral portion of the kernel source code (at /usr/src/linux/kernel/timer.c). The only unusual thing is the compiler directive `asmlinkage`, which indicates that this function is being called directly from assembly language, and thus the arguments are passed in the registers. (to be precise, `asmlinkage` is a macro which expands out to other, more confusing, gcc-specific compiler directives). If you look in the source code you won't find the exact code above, in which some of the macros have been expanded out for better readability.

Note that the uid is a property of the currently running process, and thus is fetched from the `current_task` structure via the `current` pointer.

Let's take a look at another system call (`time`) which passes an argument:

```
asmlinkage long sys_time(time_t __user * tloc)
{
        time_t i;
        struct timespec tv;

        getnstimeofday(&tv);          //Kernel keeps time in ns resolution
        i = tv.tv_sec;                          //tv_nsec ignored, therefore rounds DOWN

        if (tloc) {
                if (put_user(i,tloc))
                        i = -EFAULT;
        }
        return i;
}
```

Recall that `time` returns the time as an int, but also accepts a pointer to an int. That argument comes in to the system call as `tloc` in the code above, and if supplied, the kernel takes the value and writes it into the user's memory using the kernel function `put_user`. If the user supplied an invalid memory address, `put_user` will catch that and return a non-zero value, which will cause the system call to fail with EFAULT. We'll have a bit more to say about accessing user memory from within the kernel in a few units.

Returning from a system call

Once the system call specific handler routine `sys_XXX` returns, the system call return value is in the `%eax` register (because that's where C function return values are placed by the compiler). Looking back at the `system_call` assembly language routine, we see this return value is written to the kernel stack in the location where the `eax` register will be popped when returning back to user mode.

Now at label `syscall_exit` interrupts are temporarily masked (on multi-processor systems, this applies to the local processor only). The reason for this is to protect the next few testing and branching instructions as a *critical region*. Recall that the `thread_info` structure address is in `%ebp`. The next line of assembly code fetches the bitwise `flags` into register `%ecx` and tests to see if any flags are set which would indicate that, instead of returning directly to user mode, some other action might be required. Let's say for a moment that those flags are clear. Then the code at `restore_all` uses a macro `RESTORE_REGS` to pop all of the registers which had been saved by `SAVE_REGS`. The system call return value is now in `%eax` (regardless of how control reached `restore_all`) and the extra stacked copy of `eax` which contained the system call number is simply discarded.

Now the `iret` instruction is executed. This causes the hardware to restore the `eip, cs, eflags, esp` and `ss` registers. The result is that execution resumes in the user process, with the stack pointer back on the user's stack. All of the registers are exactly as they were when the user process executed the `INT $0x80` instruction, with the exception of the `eax` register which now holds the system call return value. The restoration of the `eflags` register means the privilege level is returned to user mode, and the interrupt mask is restored to the normal value (which when running in user mode is to allow interrupts).

Deferred return from system call

It may happen that while the process is in the kernel, something happens which makes another process potentially more runnable. In the next unit, we'll see how scheduling decisions are made. It could also happen that a signal has been posted to the process. In these cases, control does not necessarily return back to user mode after the system call.

Referring to the `entry.S` code, if the `TIF_NEED_RESCHED` flag is set, then at `work_resched` the kernel scheduler function `schedule` is called. We'll see that this function, if it picks another process to run, causes a *context switch*, and the original process appears to be frozen at the instant of having called `schedule` from `work_resched`. At some later time, the original process is selected to run again, and execution resumes at that frozen point. Control returns from `schedule`, and then the next few lines are identical to those we have already seen: they check to see if anything else has come up, or whether it is OK to return to user mode.

`work_notifysig` is used when a signal has been posted to the process, and will be discussed in a future unit.

Appendix - X86 Architecture and Assembly Language

The following material comes from ECE466 -- Compilers and is intended to assist with understand the assembly language portion of the kernel.

X86 refers broadly to a family of Intel (and compatible) microprocessors manufactured in the last 20 years or so. It is also called the X86 architecture by Intel. The first 32-bit X86 processor was the 80386. X86-64 is a 64-bit extension to X86. Intel’s is a CISC architecture which is a direct linear descendent of the very first microprocessor, the 4004 (a 4-bit product).

There are many who find the X86 architecture to be a dinosaur, and a badly designed one at that, which should have long ago become extinct. However, IBM’s choice of it for its first personal computer sealed its fate as the most popular processor architecture.

The X86-64 architecture extends the 32-bit X86 to use 64-bit registers, while retaining backwards compatibility with 32-bit X86 code.

Below is a summary of the X86/X86-64 architecture The reader is detoured to the official reference manuals for full details.

Intel vs UNIX assembly syntax

The Intel documentation uses the Intel standard assembly language syntax, but the UNIX assembler `as` follows a different convention (which is consistent across different processors). In the UNIX syntax, an identifier is unambiguously an assembler symbol. To reference a register, its name is prefixed with a percent sign, e.g. %eax. To use a symbol as an immediate value, the dollar sign is used as a prefix. Otherwise the symbol means the contents of that address. Register indirect addressing modes are indicated by brackets or parentheses. UNIX assembly instructions are opcode src1,src2,dst for 3-address instructions or opcode src,dst for 2-address. (Note that Intel syntax is dst,src). We will use the UNIX syntax in these notes. Another name for this is the "AT&T" syntax, after the original authors of UNIX

In the UNIX/AT&T syntax, where a particular opcode can be performed at different precisions, that opcode receives a letter suffix: b,w,l,q for 8,16,32 and 64-bit operations respectively.

Assembly language files are denoted with a .s or .S suffix. The latter is used in the Linux kernel for files which are hand-crafted in assembly, as opposed to .s files which were generated by the compiler. Assembly language is line-break sensitive. Each line is an instruction (not including whitespace, assembler directives and comments) and each instruction must be contained on that line. The line has 2 parts: opcode operands. These fields are separated by whitespace. The label is a symbolic name given to that instruction or memory location. It is specified as `label:` and can either appear at the beginning of the line, or on a line by itself, in which case the label is attached to the next line. By stylistic convention, the opcode is never at the beginning of the line. There is least one whitespace character (typically a tab) so that the opcode is not mistaken as a label. The operands field contains 0, 1 or more operands delimited by commas, and following the addressing mode syntax explained below.

Assembler Directives / Pseudo Opcodes

Assembler directives are pseudo-opcodes that do not correspond to actual opcodes that the processor executes, but cause the assembler to modify its operation, or to emit special code or data. These include `.text` and `.data` to switch between the text and data sections of the a.out file, `.byte` to emit a single byte, `.long` to emit a 4-byte value, and `.string` to emit a nul-terminated string. This is not an exhaustive list and the reader is referred to the documentation for `as`.

X86 Register Model

When referring to X86 registers, their size is implied by a prefix. For example, there is a 32-bit register called EAX. The least significant 16 bits of that register are called AX. It is possible to refer to the least significant byte as AL and the next most significant byte as AH. In the 64-bit X86-64 instruction set, the 64-bit version of EAX would be called RAX. We will consider the 32-bit model first.

The register model of X86 is convoluted and archaic, making efficient register allocation and instruction selection a challenge. The following general-purpose registers are typically used for holding temporary values, general integer computation, etc.
• %eax: The "accumulator". Many instructions use %eax as an implied operand.
• %ebx: The "base register" (not to be confused with %ebp).
• %ecx: The "counter register".
• %edx: The "data register".
• %esi: Source register for string operations
• %edi: Destination register for string operations

The following special registers are used for control flow:
• %eip: The "instruction pointer", aka the Program Counter. At the time of instruction execution, %eip contains the address of the next instruction to be fetched. A branch

instruction modifies %eip and causes the next instruction to be fetched from that new address.
• %esp: The stack pointer.
• %ebp: Typically used in the C / assembly language convention for the stack frame "base pointer". Aka the "frame pointer".
• %eflags: The flags register. It contains the condition code flags (carry, parity, BCD adjust, zero, signed, overflow) as well as a number of flags and control bits which can only be modified when running in Kernel (Supervisor) mode, such as the interrupt enable flag and the user/supervisor privilege level.

Segmentation, Special-Purpose and Additional Registers

The X86 addressing scheme is based on an obsolete concept known as "segment/offset" addressing. In all modern operating systems, program addresses are linear, and the segmentation is basically ignored. The register model contains the registers %cs, %ds, %ss which are initialized by the kernel and should not be touched. They are what enable code, data and stack accesses to work. Additional segment registers %es, %fs and %gs are general-purpose and, because a linear addressing model is being used, could be employed as general-purpose scratch registers, subject to some restrictions as to which registers may appear in which instructions. However, both the %fs and %gs registers are used by the kernel and the standard library, and should be avoided.

We have seen additional registers such as %cr3 and %tr. These are generally only accessible when running in kernel mode, and require special opcodes to read or write. Additional registers are present for various floating point operations. The kernel does not use these at all, but is required to save and restore them when context switching between user-level tasks, if they are being used by those tasks.

On X86-64, there are additional general-purpose registers `%r8 - %r15`.

Addressing Modes

There are a number of addressing modes which are used to specify where to find or put the operands of an instruction:
• Register Direct: Specify the register name with a % prefix, e.g. %eax.
• Immediate: The immediate value must be prefixed with the dollar sign, e.g. $1
• Memory Absolute: The absolute address of the operand is specified without a prefix qualifier. E.g. `movl $1,y` moves the immediate value 1 into the memory address which is associated with the linker symbol y.
• Base-index (Register Indirect with offset): The X86 has a handy mode for accessing elements of an array. The syntax is `disp(%base,%index,scale)` . The address of the operand is computed as `addr=base+index*scale+disp`. The base and index

may be any of the general-purpose registers (eax, ebx, ecx, edx, ebp, dsi, edi, esp (not allowed as the index)). The displacement is a 32-bit absolute address. The scale factor may be 1, 2, 4 or 8. Some of these parameters may be omitted, forming simpler addressing modes. E.g. in `movl $1, (%eax)` the eax register contains a pointer to a memory location, into which the immediate value 1 is moved.

X86 is generally a 2-address architecture, meaning that one of the operands is both a source and a destination. There are many combinations of src/dst addressing modes including some odd restrictions. Generally speaking, most opcodes allow register/register, register/immediate, register/memory or immediate/memory combinations. Memory/memory is generally not allowed.

Function Calling Convention

We will discuss what the Intel documentation calls the CDECL convention for procedure calling, as that is what is used in the C/UNIX world. Other calling conventions do exist. In the X86-32 architecture, all arguments to a function are pushed on the stack, and the return value is returned in the %eax register. If the return value is 64 bits (long long), it is returned in the register pair %edx:%eax, with the %edx being the most significant 32 bits.

Recall that %esp is the stack pointer, and the stack grows towards low memory. The PUSH instruction predecrements the stack pointer, then writes the value to (%esp). Likewise, POP reads from (%esp) and then postincrements %esp. Arguments in C are pushed to the stack in right-to-left order. Therefore, just before issuing the CALL instruction, the leftmost argument is on the top of the stack. This convention allows variadic functions to work properly. The callee does not need to know in advance (at compile time) the exact number of arguments which will be pushed. It is able to retrieve the arguments left-to-right by positive offsets from %esp.

The CALL instruction pushes the value of %eip, thus on entry to a function (%esp) contains the address of the instruction to which control should return (i.e. the instruction after the CALL). The first thing any function does is set up its local stack frame. Let's look at an example:

```
f1()
{
        f2(2);
}

f2(int b)
{
int a;
        a++;
        b--;
        return 1;
}
```

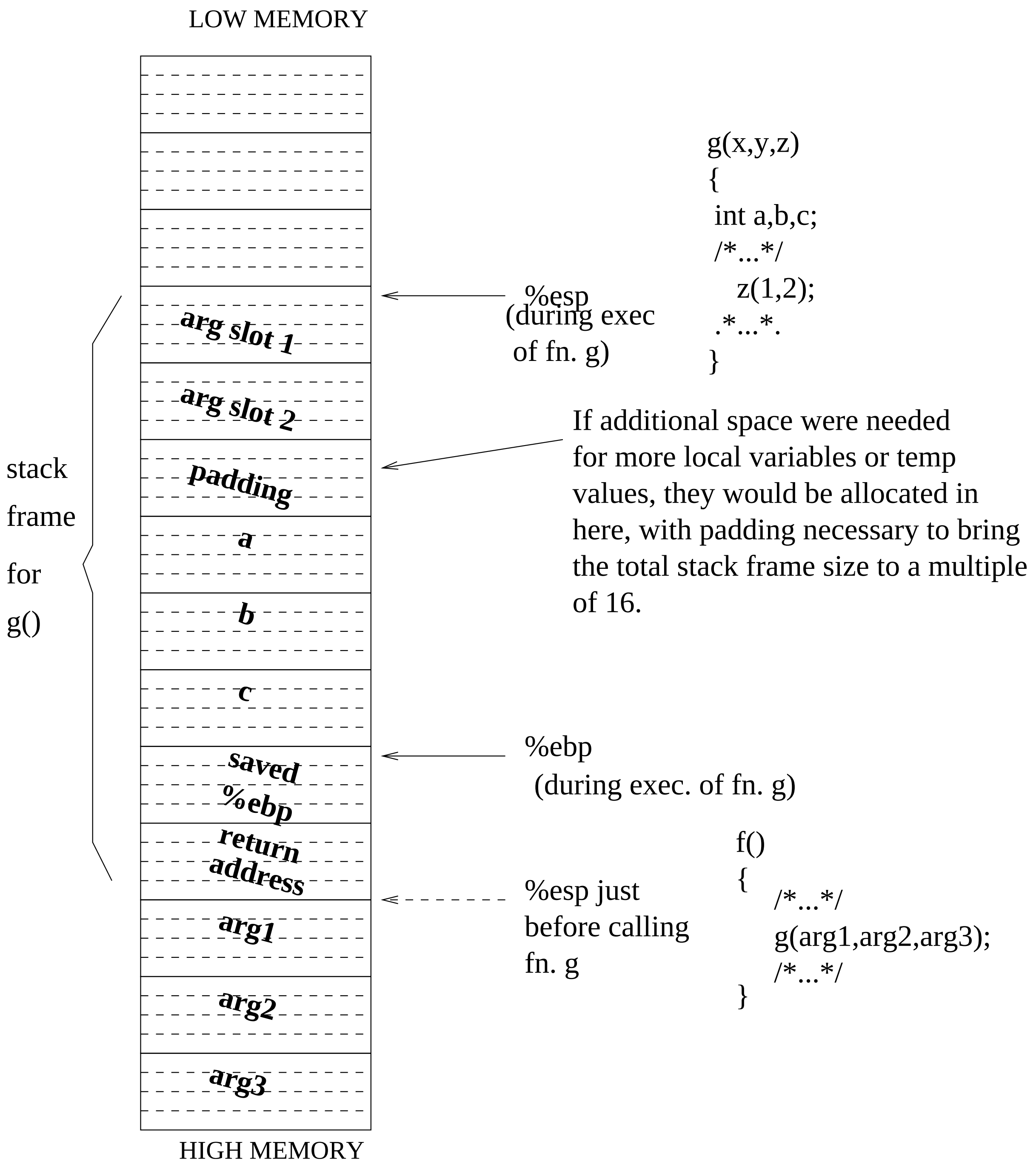
```
f1:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp                !one arg slot + one padding slot
        movl    $2, (%esp)              !put arg onto stack
        call    f2
        leave
        ret

f2:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp               !extra space for alignment
        incl    -4(%ebp)                !access local var a
        decl    8(%ebp)                 !access param b
        movl    $1,%eax                 !return value
        leave
        ret
```

The %ebp register is the frame pointer, and will be used to access both local variables and parameters. Its value must be preserved so the first action is to save it on the stack. Then the stack pointer is decremented to create room for local variables. In our example, function g has one local variable which takes up 4 bytes. The %ebp contains the value of the stack pointer after saving the old %ebp. Therefore 4(%ebp) is the return address, (%ebp) is the saved %ebp, and the first parameter is 8(%ebp). Parameters will be at positive offsets from %ebp and local variables will be at negative offsets. Generally speaking, the local variables mentioned first in a function will have the lowest memory address (i.e. highest negative offset from %ebp), but that behavior is not guaranteed.

When a function call is made, arguments can be pushed on the stack in right-to-left order, using the `pushl` instruction. After the `CALL` instruction, an `addl $X,%esp` would be needed to adjust the stack pointer and reverse the effects of the previous pushes. Alternatively, one could determine during code generation which function call (within the function being generated) has the highest number of arguments. The number of bytes thus required for passing arguments can be added to the total local stack frame size, as if these "argument slots" were hidden local variables. Then the arguments can be passed via `movl OFFSET(%esp)`, in any order desired, and there is no need to adjust the stack pointer after the call. This is the approach that gcc takes.

Upon leaving a function, the LEAVE instruction is used, which performs two operations: %ebp is moved into %esp, thus restoring the stack pointer to its value just after the base pointer save on entry, then %ebp is popped from the stack. Now everything is restored, and the RET instruction pops the return address from the stack and resumes execution in the caller.

If the compiler chose to use any registers which are callee-saves, we would see pushes of those registers on entry and corresponding pops on exit.


LOW MEMORY
arg slot 1
arg slot 2
padding
a
b
c
saved %ebp
return address
arg1
arg2
arg3
HIGH MEMORY
stack frame for g()
%esp (during exec of fn. g)
If additional space were needed for more local variables or temp values, they would be allocated in here, with padding necessary to bring the total stack frame size to a multiple of 16.
%ebp (during exec. of fn. g)
%esp just before calling fn. g

```
g(x,y,z)
{
 int a,b,c;
 /*...*/
   z(1,2);
 .*...*.
}
```

```
f()
{   /*...*/
    g(arg1,arg2,arg3);
    /*...*/
}
```

X86-64 Function Calling

Under the 64 bit architecture, the first 6 integer arguments are passed in registers, rather than on the stack. Arguments are placed in left-to-right order in registers %rdi, %rsi, %rdx, %rcx, %r8, %r9. If there are additional arguments, they are put on the stack right-to-left, i.e. with the right-most argument at the highest memory address, just like X86-32.

If structs are passed as arguments, they are always placed on the stack. The integer return value is in the %rax register.

This hybrid register/memory argument passing model introduces some complexity with variadic functions, aka `<stdarg.h>`. GCC implements `stdarg` as a compiler built-in.

X86-64 Global Variables

There is an odd limitation in the X86-64 instruction set: the absolute addressing mode is not supported for 64-bit addresses. To access a memory operand, a register indirect addressing mode must be used.

```
extern int i;

f()
{
         i=2;
}

f:
        pushq   %rbp                            #Prologue, save base pointer
        movq    %rsp, %rbp            #Set new base pointer
          subq     $32, %rsp          #Create stack frame
        movl    $2, i(%rip)           #Program Counter Relative mode
        leave
        ret
```

There will be a 32-bit "hole" in the `movl` opcode which will be a program counter relative relocation type (similar to the example of the CALL opcode earlier in this unit). At link time, when the address of symbol i has been resolved, this hole will be filled with the i's address, minus the address of the hole itself.

This introduces a limitation that code and data must fall within the same contiguous 2GB memory region at run time, which the X64-64 spec calls a "medium" memory model. To use a "large" memory model where code and data may be anyplace within the 64-bit address space, different opcodes are used:

```
          movabsq   $i, %rax          #Move 64 bit immediate value to rax
        movl    $2, (%rax)            #Register indirect
```

Caller/Callee saves

It is the case for any architecture and operating system that there is a function calling "convention" which specifies how arguments are passed and returned, and how registers may be used. This convention dictates which of the registers are expected to survive a function call, and which ones may be used as "scratch" registers, and are therefore expected to be volatile across function calls. Another way of saying this is there are caller-saved registers (the scratch registers.. if the caller wants to keep a value in there through a function call it must explicitly save it) and callee-saved registers (if a function wants to use one of these registers it must explicitly save it on entry and restore it before returning).

In the X86-32 architecture under UNIX, the %eax,%ecx,and %edx registers are scratch registers (caller-saves). You will find that the compiler tends to put short-lived values in these registers. Of course the %eflags register is also expected to be modified by a function call. The %ebx,%edi,%esi and %es [CAUTION: this is a 16-bit register] registers are callee-saved. The compiler may use these for longer-lived values (such as local variables which are assigned to a register for all or part of the function to improve speed). However, if one of these registers is used by the compiler, it must emit code to push it on the stack on entry, and pop it on return.

On X86-64, the caller-save (scratch) registers are %rax,%rcx,%rdx,%rsi,%rdi, and %r8-%r11, while the callee-save (long-term) registers are %rbx, %r12-%r15. Note that %rsi and %rdi are caller-save on 64 bit, whereas they were callee-save on 32-bit. This is because they are used for argument passing on 64 bit.

X86 General-Purpose Register Summary

-32 reg	Saved by	Notes	-64 reg	Saved by	Notes
%eax	Caller	fn retval	%rax	Caller	fn retval
%ebx	CallEE		%rbx	CallEE	
%ecx	Caller		%rcx	Caller	arg #4
%edx	Caller	longlong ret	%rdx	Caller	arg #3
%edi	CallEE		%rdi	Caller	arg #1
%esi	CallEE		%rsi	Caller	arg #2
%es	CallEE		N/A		
			%r8	Caller	arg #5
			%r9	Caller	arg #6
			%r10	Caller	
			%r11	Caller	
			%r12	CallEE	
			%r13	CallEE	
			%r14	CallEE	
			%r15	CallEE	