Signals

From the user-level standpoint signals appear to be lightning bolts which can come out of the sky at any moment. In fact, signals are carefully controlled and manipulated by the kernel. There are two distinct phases to a signal:

• **Raising** aka **Generating** a signal: The kernel keeps track of pending signals for each process. There are numerous kernel routines which generate a new signal. These are called either by a system call (e.g. `kill`) or as part of the kernel's event processing (e.g. handling a synchronous fault interrupt).

• **Delivering** the signal: The signal is "noticed" by the process in kernel mode as it is about to return back to user mode. Instead of returning directly back to the user program, the appropriate action is taken. This action may be to terminate the process, terminate and dump core, or invoke the defined signal handler.

Signal data structures

In the Linux kernel, the `struct task_struct` of each process contains several fields pertaining to signals:

```
struct task_struct {
/*...*/
struct signal_struct         *signal;  /* shared signals */
struct sigpending       *pending;     /* private signals */
struct sighand_struct   *sighand;     /* signal handling */
sigset_t                blocked;      /* bitmask of blocked signals */
```

There are two lists of pending signals, and the distinction comes into play with multi-threaded programs. Shared signals are those sent to the entire thread group. The signal will be delivered once, to one of the threads (which is not currently blocking that signal number). Private signals are sent to specific threads. We will ignore this aspect of signals as well as POSIX real-time signals, and review only traditional, shared signals.

The `struct signal_struct` contains some things which really have nothing to do with signals, but are there for historical reasons:

```
struct signal_struct {
/*...*/
wait_queue_head_t      wait_chldexit; /* Wait queue for wait syscall */
struct sigpending      shared_pending;  /* Pending signal list */
int                    group_exit_code; /* Process termination code */
```

Of interest is `shared_pending`, which is the list of shared signals (i.e. traditional signals which are delivered to the entire process and not a particular thread). This structure is:

```
struct sigpending {
        struct list_head list;                 /* list of sigqueue */
        sigset_t  signal;            /* bitmask */
};
```

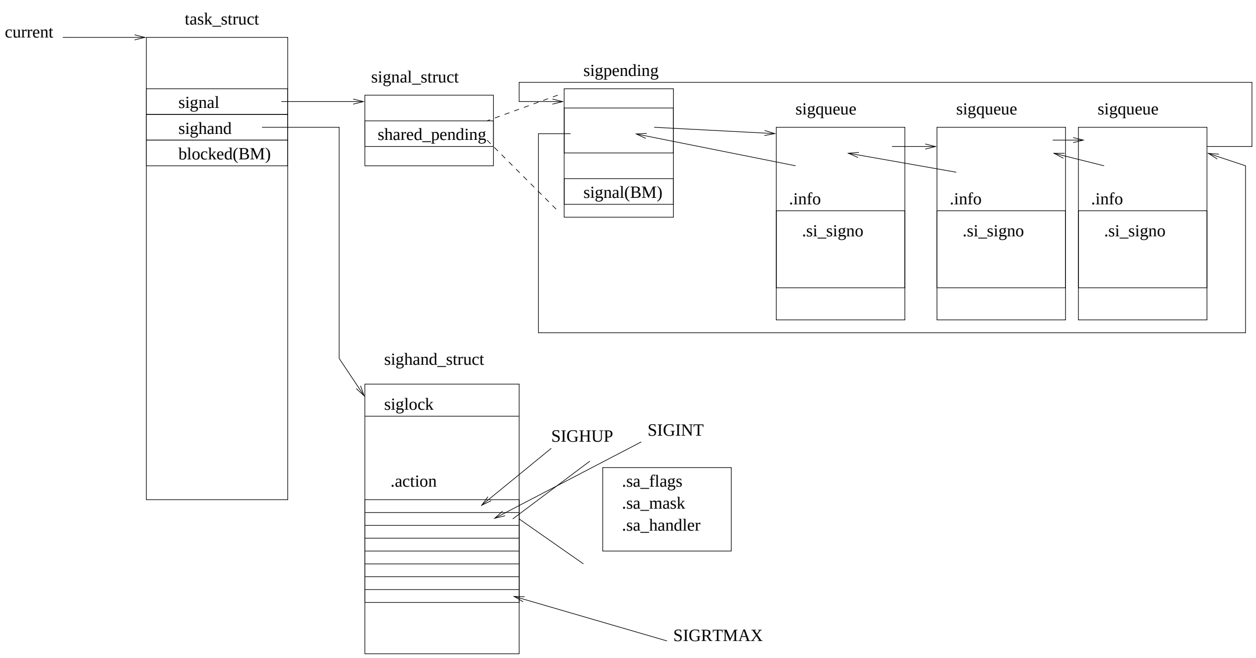
Therefore the list of pending signals for the process is rooted at `current->signal->shared_pending.list`. The bitmask allows a quick determination if a particular signal number (e.g. SIGINT) is pending. Then each member of the list is:

```
struct sigqueue {
         struct list_head  list;         /* chain pointers */
         spinlock_t        *lock;        /* ptr to lock in sigaction */
         int               flags;
         siginfo_t         info;              /* sig# and other info */
         struct user_struct *user;    /* ptr back to user struct */
};
```

The `struct sighand_struct *sighand` member of the `task_struct` contains the signal handling information for the task:

```
struct sighand_struct {
        atomic_t                count;          // don't worry about it!
        struct k_sigaction      action[_NSIG];  //just like struct sigaction
        spinlock_t              siglock;
        wait_queue_head_t       signalfd_wqh;            // don't worry about it!
};
```

We see that there is a table, with one entry for each of the 64 possible signal numbers, giving the handling status of each, represented as a `struct k_sigaction`, which in the case of Linux on a 32-bit X86 machine is simply the same as the user-level `struct sigaction`. Each entry thereof contains a user-mode handler address (which may also contain the value SIG_DFL or SIG_IGN), a set of flags, and a bitmask of additional signals to be masked when handling this signal.



Generating a signal

A signal may be generated or raised against a target task :
• When another (or possibly the same) task explicitly sends the signal, e.g. with the kill system call.
• When another process causes an event which generates a signal. E.g. a child process exits and generates SIGCHLD for its parent.
• When the target task is in the kernel for a system call, and a condition is encountered which causes a signal. E.g. SIGPIPE.
• When the target task is in the kernel for a fault handler and the kernel determines that the fault merits a signal. E.g. SIGSEGV, SIGBUS, SIGILL.
• As a direct or indirect result of a kernel hardware interrupt handler. Examples: I/O event completes and process has registered to receive SIGIO signals. Character arrives on a terminal device which is the interrupt character (typically Control-C), which generates a SIGINT to the foreground processes attached to that terminal. Timer expires and process has requested SIGALARM or SIGVTALRM.

Following a kill

Generation of a signal is the same, regardless of the cause. Let us follow the chain of events when one process sends a signal to another with the `kill` system call:

```
asmlinkage long
sys_kill(int pid, int sig)
{
        struct siginfo info;

        info.si_signo = sig;
        info.si_errno = 0;
        info.si_code = SI_USER;
        info.si_pid = current->tgid;
        info.si_uid = current->uid;

        return kill_something_info(sig, &info, pid);
}
```

The `siginfo` structure stores information about a particular instance of a signal. It is a rather lengthy definition because it contains a union of all the possible sets of information which could accompany a signal, depending on the source (kill, fault, child exit, etc.). The interested reader is invited to look at `/usr/src/linux/asm-generic/siginfo.h`. In this case, the code `SI_USER` indicates the signal originated with another user-mode process.

`kill_something_info` interprets the `pid` parameter (recall that in addition to explicitly giving the pid of the target process, one can target an entire *process group* or all processes belonging to the same user). It in turn calls:

```
int kill_proc_info(int sig, struct siginfo *info, pid_t pid)
```

```
{
        int error;
        struct task_struct *p;

          // Searching task list done under blocking reader lock
          // so we don't either miss a task or select a task which
          // has already disappeared
        read_lock(&tasklist_lock);
        p = find_task_by_pid(pid);
        error = -ESRCH;                // Error if pid not found
        if (p)
                error = group_send_sig_info(sig, info, p);
        read_unlock(&tasklist_lock);
        return error;
}
```

Here we see the *task list* in action. There are a number of identifiers which can be used to find one or more processes. They are:

• PID: The real task (process) id, unique for each task in the system.

• TGID: Thread group id. Equivalent to PID for single-threaded applications. For multi-threaded programs, there are multiple PIDS per TGID.

• PGRP: Process group id. Process groups are used primarily for signal delivery targeting where a group of processes is working together, e.g. a pipeline.

• SID: Session id. All processes associated with a particular interactive login session share an SID. This is used in conjunction with the tty subsystem, e.g. to figure out which processes should be killed when the session is lost.

Besides the global list of all tasks (`tasks` list_head entry in struct task_struct), there are multiple hash lists used to quickly find matching `struct task_struct` based on a given id (`struct pid_link pids[PIDTYPE_MAX]` in `struct task_struct`). So, above we see `find_task_by_pid` being used to look up the pid and return the task structure pointer. Now `group_send_sig_info` will be used to post a shared signal to the entire thread group (again, for conventional single-threaded applications, this is just the single target process).

```
int group_send_sig_info(int sig, struct siginfo *info, struct task_struct *p)
{
        unsigned long flags;
        int ret;

        ret = check_kill_permission(sig, info, p);
        if (!ret && sig) {
                spin_lock(&p->sighand->siglock, flags);
                ret = send_signal(sig, info, p);
                spin_unlock(&p->sighand->siglock, flags);
        }

        return ret;
}
```

```
/* Greatly simplified to remove details regarding per-thread and rt signals */
int send_signal(int sig, struct siginfo *info, struct task_struct *t)
{
 struct sigqueue *q;
         trace_signal_generate(sig,info,t);     //hook for strace
        if (sig_ignored(p, sig))      /* check sigaction array  for SIG_IGN */
         {
                  /* sig_ignored concludes that the signal can be ignored
                     only if all of the following 3 conditions are true: */
                           1) The process is not being traced (if so
                                  signal must be taken so tracing process
                                  will see this )
                           2) The sa_handler in the sighand->action table
                                  is SIG_IGN
                           3) The signal is not being blocked (if it were,
                                  we must remember it....when it becomes
                                  unblocked the disposition may no longer
                                  be SIG_IGN)                              */
               return 0;
         }
         /* Non real-time signals (<32) do not queue more than once */
         if (sig<32 && sigismember(&p->pending.signal)
                  return 0;
         q=sigqueue_alloc(sig,t);
         if (q) {  //if q alloc fails, usually silent loss of info
                  list_add_tail(&q->list,&t->signal->shared_pending.list);
                  copy_siginfo(&q->info,info);
         }
         sigaddset(&t->signal->shared_pending.signal,sig);
         complete_signal(sig,t);
         return 0;
}

complete_signal(int sig, struct task_struct *p)
{
         struct task_struct *t;
         //for multithreaded processes, find best thread to take signal
         //for singlethreaded, t=p
         // call signal_wake_up(t), code inlined below:
         t->thread_info.flags |= TIF_SIGPENDING;
         if (t->state==TASK_INTERRUPTIBLE)
                  wake_up(t);
         else
                  kick_process(t);   // Will send an IPI on multiproc system

}
```

`check_kill_permissions` verifies that the signal number is valid (between 0 and 63) and that the sending user has permission to send it to the target process. The signal number 0 is used just to test for such permission (or to probe for the existence of a particular pid) and does not actually cause a signal to be posted.

The process which sent the signal will see a successful return from `sys_kill`. The target process has the `TIF_SIGPENDING` flag set. Therefore, the next time it is about to return from kernel to user mode, this flag will be noticed, and rather than returning, the signal will be delivered.

On a uniprocessor system, the signal sender task and the target task (unless they are the same) can not be truly executing at the same time. Since the signal sender is running in the kernel, the target is either sleeping or in the ready (not on cpu) state. But, on a multi-processor system, it could be the case that the target is also running at the time that the sender is in the kernel sending the signal. Since signals can only be noticed (delivered) when control is about to return from kernel mode back to user mode, some mechanism must be present to force the target task to enter kernel mode (other than the timer tick interrupt, which may be a very long millisecond away). This is handled in complete_signal() by the kick_process function, which can send an Inter-Processor Interrupt (IPI) to the target task's processor.

Signal Delivery

Signals may be generated against a task at any time, but **can only be delivered to the task when it is running in the kernel, and about to return back to user mode.**

When the assembly language glue code dealing with return to user mode (see unit 8) sees flags in the thread_info structure, control passes to `work_notify_sig` which in turn calls the kernel function `do_notify_resume:`, which calls `do_signal` (code below). This picks a signal to deliver (there may be multiple pending signals) and delivers it. A signal which is currently blocked is never delivered. There are three basic outcomes to the signal delivery mechanism invoked under `do_signal`: ignore, terminate or handle.

A signal may be deliverable to the process for which the disposition in the sigaction table is SIG_IGN. This can happen if the signal was previously sent while that signal number was blocked, and now the signal mask has been changed to unblock it. A signal may also be set for SIG_DFL but the default action defined by the kernel for that signal number is to ignore the signal (this is the case for SIGCHLD, SIGURG, SIGWINCH and SIGPWR). When do_signal determines that the signal is ignorable, it simply discards the associated `sigqueue`.

The default action for most signals is to terminate, possibly with a core dump. If core dump is needed, `do_coredump` is called which creates a file called `core` in the process's current working directory and writes to that file the contents of all valid memory regions in the process address space. The coredump can be used for post-mortem analysis. Because `do_coredump` runs synchronously, it may put the process to sleep waiting for write I/O operations to conclude. It also checks for file permissions as if `core` was being opened using O_CREAT|O_WRONLY|O_TRUNC. Once the core dump completes (if applicable) then do_group_exit is called which terminates the current process with the signal number as the exit reason.

The final possibility is to handle the signal, which is discussed following the code.

```
#From kernel/entry.S
work_notifysig:                                 # thread flags already in ECX
        movl %esp, %eax                         # put sp into arg 1
        xorl %edx, %edx                         # put NULL into arg 2
        call do_notify_resume
        jmp resume_userspace          # resume return to userland

// From kernel/signal.c
// Note that regs points to the saved user-mode registers on the kernel stack
__attribute__((regparm(3)))
void do_notify_resume(struct pt_regs *regs, sigset_t *oldset,
                      __u32 thread_info_flags)
{
        /* Pending single-step? */
        if (thread_info_flags & _TIF_SINGLESTEP) {
                regs->eflags |= TF_MASK;
                clear_thread_flag(TIF_SINGLESTEP);
        }
        /* deal with pending signal delivery */
        if (thread_info_flags & _TIF_SIGPENDING)
                do_signal(regs,oldset);

        clear_thread_flag(TIF_IRET);
}

int fastcall do_signal(struct pt_regs *regs, sigset_t *oldset)
{
        siginfo_t info;
        int signr;
        struct k_sigaction ka;

          /* We could be returning back to kernel mode from an interrupt
                   handler.  If so, we don’t do anything now.  We only
                   handle signals when about to return to user mode */
        if (!user_mode(regs)) return 1;

        if (!oldset) oldset = &current->blocked;

             // Find a signal to handle, or return 0 if none.
             // Could be that we woke up on a signal which turned out
             // to be one that we ignore
        signr = get_signal_to_deliver(&info, &ka, regs, NULL);
        if (signr > 0) {
                return handle_signal(signr, &info, &ka, oldset, regs);
        }
           /* Recall that the orig_eax slot contains the original value */
           /* of the eax register when we first entered kernel mode.  If */
           /* it is non-zero, then we came in as a system call and the   */
           /* saved value is the original system call number.  The      */
           /* regs->eax slot contains the return value from the system call */
```

```
        if (regs->orig_eax >= 0) {
           /* If we arrive here, we are returning to user mode from */
           /* an interrupted system call, but the signal is being   */
           /* ignored, not handled.                                  */
           /* Similar code in handle_signal.  See text.             */
                if (regs->eax == -ERESTARTNOHAND ||
                    regs->eax == -ERESTARTSYS ||
                    regs->eax == -ERESTARTNOINTR) {
                        regs->eax = regs->orig_eax;
                        regs->eip -= 2;
                }
                   // Restart blocks are used when restarting the system
                   // call is more complicated than just calling it again.
                   // They force re-executing of a special "restart" system
                   // call which effects the more complex semantics.

                if (regs->eax == -ERESTART_RESTARTBLOCK){
                        regs->eax = __NR_restart_syscall;
                        regs->eip -= 2;
                }
        }
        return 0;
}

int get_signal_to_deliver(siginfo_t *info, struct k_sigaction *return_ka,
                          struct pt_regs *regs, void *cookie)
{
        sigset_t *mask = &current->blocked;
        int signr = 0;

        spin_lock_irq(&current->sighand->siglock);
        for (;;) {           // For each queued signal
           struct k_sigaction *ka;
                   /* dequeue_signal examines the signal queue in numerical  */
                   /* order via the bitmap.  It then removes that entry from */
                   /* the signal queue and returns the signal number.  Signals */
                   /* which are currently blocked are never selected         */
              signr = dequeue_signal(current, mask, info);

              if (!signr) break; /* no pending signals, will return 0 */
              ka = &current->sighand->action[signr-1];
              if (ka->sa.sa_handler == SIG_IGN)
                        continue;  /* Try next pending signal */
              if (ka->sa.sa_handler != SIG_DFL) {
                        *return_ka = *ka;

                        if (ka->sa.sa_flags & SA_ONESHOT)
                                ka->sa.sa_handler = SIG_DFL;

                        break; /* will return non-zero "signr" value */
              }
                   /* If we get here, the handling must be SIG_DFL */
              if (sig_kernel_ignore(signr)) // If default for this sig
```

```
                        continue;           // is to ignore

                /* init process (pid==1) gets no signals it doesn't want.  */
                if (current->pid == 1)
                        continue;
              spin_unlock_irq(&current->sighand->siglock);
                /* Default action must be to terminate */
              current->flags |= PF_SIGNALED;
                   /* If coredump is associated with this signr, do it */
              if (sig_kernel_coredump(signr))
                     do_coredump((long)signr, signr, regs);

              do_group_exit(signr);     // Exit with signr as exit code
                /* NOTREACHED */
        }
        spin_unlock_irq(&current->sighand->siglock);
        return signr;
}


static int
handle_signal(unsigned long sig, siginfo_t *info, struct k_sigaction *ka,
              sigset_t *oldset, struct pt_regs * regs)
{
        int ret;

        /* Are we returning from an interrupted system call? */
        if (regs->orig_eax >= 0) {
                /* If so, check system call restarting.. */
                switch (regs->eax) {            /* system call return value */
                        case -ERESTART_RESTARTBLOCK:
                        case -ERESTARTNOHAND:
                                regs->eax = -EINTR;
                                break;
                        case -ERESTARTSYS:
                                if (!(ka->sa.sa_flags & SA_RESTART)) {
                                        regs->eax = -EINTR;
                                        break;
                                }
                        /* fallthrough */
                        case -ERESTARTNOINTR:
                                regs->eax = regs->orig_eax;
                                regs->eip -= 2;
                }
        }
           /* Set up the expected type of stack frame */
        if (ka->sa.sa_flags & SA_SIGINFO)
                ret = setup_rt_frame(sig, ka, info, oldset, regs);
        else
                ret = setup_frame(sig, ka, oldset, regs);

        if (ret) {
                spin_lock_irq(&current->sighand->siglock);
```

```
                sigorsets(&current->blocked,&current->blocked,&ka->sa.sa_mask);
                if (!(ka->sa.sa_flags & SA_NODEFER))
                        sigaddset(&current->blocked,sig);
                recalc_sigpending();   // Clear TIF_SIGPENDING if no more sigs
                spin_unlock_irq(&current->sighand->siglock);
        }

        return ret;
}
```

Setting up the stack frame to handle signal

When a signal handler is to be invoked, `setup_frame` pushes the following onto the **user mode stack**:

```
struct sigframe {
   void *pretcode;                    /* return address after handler */
   int  sig;                                    /* signal number */
   struct sigcontext sc;              /* hardware context and blocked sigs */
   struct _fpstate fpstate;             /* used for floating point context*/
   char retcode[8];                     /* historical */
}
```

The current hardware context (stored in the kernel stack and accessible via the `regs` local variable) is copied into the `struct sigcontext` on the user-mode stack. This structure also contains the bit vector of signals which were being blocked before the handler was invokved (recall that the signal being handled is generally added to the set of blocked signals, plus any additional signals specified with the `sigaction` system call). The kernel stack is then modified so that upon return to user mode, the stack pointer will be the new top of the user-mode stack (with the `sigframe` on the top of the stack), the %eip register will be the address of the signal handler routine, the %eax register will contain the signal number.

Now control returns to user mode at the address of the handler function. From the standpoint of the handler, it has been called from the point in the user's code where the signal was received, as if by a function call, and the signal number is its first argument. (If the `SA_SIGINFO` flag was specified in the `sigaction` for this signal, then a more elaborate stack frame is created which has the 3 arguments: signal number, siginfo, context).

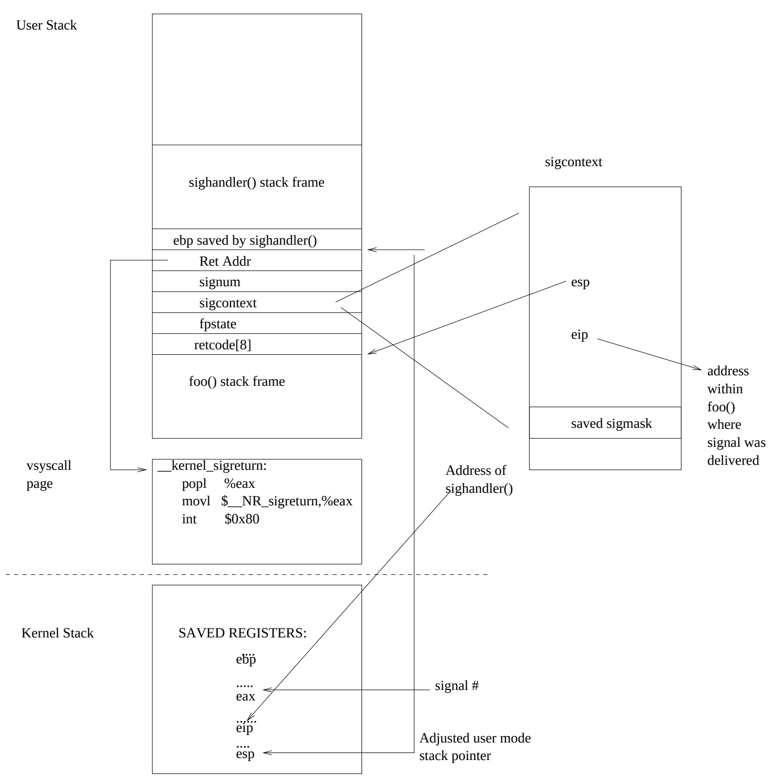
The handler is of course free to call `longjmp` in which case the stack frame which the kernel carefully crafted will simply become irrelevant. However, let us assume that the handler returns. It finds a special return address on the stack. This points to a small area of memory which the kernel creates in the user-mode address space of each process, called the *vsyscall* page (also known as the `vdso` region). This page is used for a number of system call interfacing issues. Here the process finds a few carefully chosen instructions:

```
__kernel_sigreturn:
```

```
        popl    %eax                            #just discard signal #
        movl    $__NR_sigreturn, %eax           #syscall #
        int     $0x80
```

Control now re-enters the kernel with a special system call `sys_sigreturn`, which takes the user-mode hardware context which was saved on the *user-mode stack* and copies it back to the *kernel-mode stack*, where it will be restored to the user-mode registers upon return to user mode again. The bit vector representing signals which were being blocked prior to the handler is also read out of the user-mode stack and restored as the current set of blocked signals. The signal handling frame which had been pushed on the user-mode stack is now discarded. At this point, the user-mode %esp saved on the kernel mode stack is once again the original value, as is the %eip register (but see below about restarted system calls), so now when control leaves the `sys_sigreturn` system call and returns back to user mode, execution will resume exactly where it left off. Of course, if there are additional pending signals, the `TIF_SIGPENDING` flag will still be on, and these signals will be dealt with now.

The figure below depicts a process which has entered kernel mode while executing in user mode in the function `foo()`. This entry may have been via a system call, a fault, or a hardware interrupt. A signal is about to be delivered to this process, which has specified the handler `signalhandler()` for the signal number in question. We see the kernel stack and the user-mode stack, with the stack frame which the kernel has crafted, as control is about to exit the kernel and return to user mode.



sigsuspend

Let's look at the sigsuspend system call which came in handy for problem set #7:

```
asmlinkage int
sys_sigsuspend(int history0, int history1, old_sigset_t mask)
{
        struct pt_regs * regs = (struct pt_regs *) &history0;
        sigset_t saveset;

        mask &= _BLOCKABLE;            //e.g. can't block sig#9
        spin_lock_irq(&current->sighand->siglock);
        saveset = current->blocked;
        siginitset(&current->blocked, mask);
        recalc_sigpending();
        spin_unlock_irq(&current->sighand->siglock);

        regs->eax = -EINTR;
        while (1) {
                current->state = TASK_INTERRUPTIBLE;
                schedule();
                if (do_signal(regs, &saveset))
                        return -EINTR;
        }
}
```

This is fairly sloppy code in that `history0` and `history1` are defined as place-holder variables to locate the saved register values which are on the kernel stack. Also note that while from the user-level C library the signal mask parameter is passed as a pointer, the library wrapper code has already dereferenced this pointer and passed the contents in to the system call.

Note that the signal mask is changed under protection of a mutex, and then with that mutex still held `recalc_sigpending` is called which re-evaluates the list of pending signals to see if any of them have now become un-blocked. If so, the `TIF_SIGPENDING` flag gets set. Now the system call puts the caller into an interruptible sleep. When a signal arrives and is successfully delivered (`do_signal` returns a non-zero value) then `sigsuspend` returns with error `EINTR` (this particular system call always returns an error...if the signal which woke the process up is handled by a signal handler which then does a `longjmp`, then the system call does not appear to return at all).

Restarting system calls

When a signal is posted to a process which is sleeping in a system call (e.g. waiting for a character to satisfy a read system call) in the TASK_INTERRUPTIBLE state, this causes the process to wake up. The system call will then fail with one of a number of error codes which, in the Linux kernel, are internal to the kernel and dictate if and how the system call may be restarted. As an example, look at the `pipe_read` code in unit 9.

Note that when the `pipe_wait` returns, if there are any pending signals, the read system call returns immediately with `ERESTARTSYS`. System calls such as read and write are good candidates for seamless restart, because they are sequential.

When the system call returns and control is about to go back to user mode, the signal is noticed. What happens next depends on the internal error code which the system call routine returned:

• EINTR: This system call can not be restarted. The error EINTR will be returned to the user.

• ERESTARTSYS: The system call will be restarted if the disposition of the signal in question is DFL or IGN. If there is a handler defined, the system call will only by restarted if the SA_RESTART flag is set for that signal number (e.g. through the `sigaction` system call). System call restart will happen after the handler returns. Most system calls use ERESTARTSYS.

• ERESTARTNOINTR: The system call will always be restarted, even if the handler does not have SA_RESTART set.

• ERESTARTNOHAND: The system call will be restarted if the signal disposition is DFL or IGN. If there is a handler, the system call will not be restarted, regardless of SA_RESTART, and upon completion of the handler EINTR will be returned from the system call.

• ERESTART_RESTARTBLOCK: This is a special case used for a few time-related system calls which require specific code to be executed prior to system call restart.

When it has been decided that the system call will be restarted, the `regs->eip` value is decremented by two bytes, and the `regs->eax` slot is set to `regs->orig_eax` which is the value that the eax register had upon entry (i.e. the original system call number). If there is no signal handler, then control will immediately resume in user mode, but instead of executing the **next** instruction **after** the system call (which is a two-byte instruction `INT $0x80`,) that same system call opcode will be re-executed, and since the original system call number has been restored to eax, the same system call will be made again. If there is a handler, it will be executed as described above, and assuming that it returns (as opposed to calling `longjmp`) it will wind up calling the `sigreturn` system call. This will restore the %eip value which had previously been decremented by two bytes before the handler was invoked, and so now when execution resumes after the handler, the system call will be re-executed.

If restarted system calls seem like a kluge, it is because they are indeed. But there is little choice. Once control leaves the kernel and a signal handler is invoked in the user program, and that handler returns, then something must be done if the signal handler is to appear to be seamless. In most cases, an EINTR error return from the interrupted system call is a spurious error, and makes systems programming difficult. Most system calls are designed to be restartable, but some, because of their semantics, can not be sensibly restarted from scratch. The systems programmer is cautioned to read documentation carefully for each system call.