Signals

Have a look

- Before getting into **Signals**, please take a look at
  - LK_Bird's Eye View sessions,

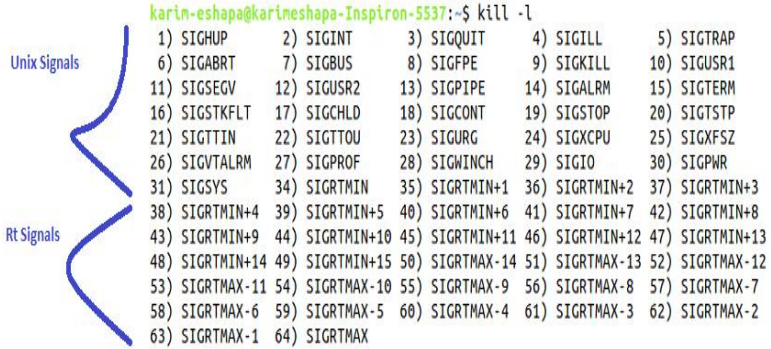
    S7 Process Scheduler,Time,IPC.

Signals

- Signals
  - Signals are short messages delivered to a process or a process group.
  - Linux categorizes signals into 2 groups, general-purpose POSIX (classic Unix signals) and real-time signals.
    Each group consists of **32** distinct signals.
  - General-purpose category are bound to a specific system event, but **Rt** category aren't bound to a specific event, and are free for applications to engage for process communication.

    The main difference that **Rt** signals of the same kind may be queued. This ensures that multiple signals sent will be received.
  - Generally there is a list of actions that a process can set up as its signal handling,

    1) Kernel handler.
    2) Process defined handler.
    3) Ignore.



/include/uapi/asm-generic/signal.h

```
#define _NSIG       64
#define _NSIG_BPW   __BITS_PER_LONG
#define _NSIG_WORDS (_NSIG / _NSIG_BPW)

#define SIGHUP       1
#define SIGINT       2
#define SIGQUIT      3
#define SIGILL       4
#define SIGTRAP      5
...

#define SIGRTMIN    32
#ifndef SIGRTMAX
#define SIGRTMAX    _NSIG
#endif
```

Signals

- Signals, *Cont'd*
  - List of actions, *Cont'd*

    1) Kernel handler :
    The kernel implements a default handler for each signal.Default handler routines can be,
    - *Ignore*: Nothing happens.

    - *Terminate*: Kill the process.

    - *Stop*: Stop all the threads in the group.

    - *Coredump*: Write a core dump file describing all threads using the same ***mm (Process Address Space)*** and then kill the threads.

    2) Process defined handler: A process is allowed to implement its own signal handlers.

    3) Ignore: A process is also allowed to ignore the occurrence of a signal, but it needs to announce its intent to ignore by invoking the appropriate system call.

Default Actions for Standard Signals

Action	Signals
Ignore	SIGCONT, SIGCHLD, SIGWINCH, SIGURG
Terminate	SIGHUP, SIGINT, SIGKILL, SIGUSR1, SIGUSR2, SIGALRM, SIGTERM, SIGVTALRM, SIGPROF, SIGPOLL, SIGIO, SIGPWR and all real-time signals.
Stop	SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU
Core dump	SIGQUIT, SIGILL, SIGTRAP, SIGABRT, SIGBUS, SIGFPE, SIGSEGV, SIGXCPU, SIGXFSZ, SIGSYS, SIGXCPU, SIGEMT

Signals

- User-mode Signal APIs
  - You can find all user-space APIs definitions in ***glibc,*** Bootlin link ***: GLIBC***

  - ***sigaction()*** : User-mode processes use this POSIX API to examine or change the handler of a signal.

    int sigaction(int signum, const struct sigaction *restrict act, struct sigaction *restrict oldact);

    signum : specifies the signal and can be any valid signal.

    act, oldact : If ***act*** is non-NULL, the new action for signal signum is installed from act. If ***oldact*** is non-NULL, the previous action is saved in ***oldact***.

    struct sigaction{}
    sa_handler specifies the action to be associated with signum.

    sa_flags : specifies a set of flags which modify the behavior of the signal.

man2 sa_flags

SA_NOCLDSTOP
If *signum* is **SIGCHLD**, do not receive notification when child processes stop (i.e., when they receive one of **SIGSTOP**, **SIGTSTP**, **SIGTTIN**, or **SIGTTOU**) or resume (i.e., they receive **SIGCONT**) (see wait(2)). This flag is meaningful only when establishing a handler for **SIGCHLD**.

SA_NOCLDWAIT (since Linux 2.6)
If *signum* is **SIGCHLD**, do not transform children into zombies when they terminate. See also waitpid(2). This flag is meaningful only when establishing a handler for **SIGCHLD**, or when setting that signal's disposition to **SIG_DFL**.

If the **SA_NOCLDWAIT** flag is set when establishing a handler for **SIGCHLD**, POSIX.1 leaves it unspecified whether a **SIGCHLD** signal is generated when a child process terminates. On Linux, a **SIGCHLD** signal is generated in this case; on some other implementations, it is not.

SA_NODEFER
Do not add the signal to the thread's signal mask while the handler is executing, unless the signal is specified in *act.sa_mask*. Consequently, a further instance of the signal may be delivered to the thread while it is executing the handler. This flag is meaningful only when establishing a signal handler.

SA_NOMASK is an obsolete, nonstandard synonym for this flag.

SA_ONSTACK
Call the signal handler on an alternate signal stack provided by sigaltstack(2). If an alternate stack is not available, the default stack will be used. This flag is meaningful only when establishing a signal handler.

SA_RESETHAND
Restore the signal action to the default upon entry to the signal handler. This flag is meaningful only when establishing a signal handler.

SA_ONESHOT is an obsolete, nonstandard synonym for this flag.

Signals

- User-mode Signal APIs, *Cont'd*
  - ***sigprocmask()*** : allow to block or unblock signal delivery.

    int sigprocmask(int how, const sigset_t *restrict set, sigset_t *restrict oldset);

    The behavior of the function call is dependent on the value of ***how***, as follows,

    SIG_BLOCK : The set of blocked signals is the union of the current set and the ***set*** argument.

    SIG_UNBLOCK : The signals in ***set*** are removed from the current set of blocked signals. It is permissible to attempt to unblock a signal which is not blocked.

    SIG_SETMASK : The set of blocked signals is set to the argument ***set***.

Signals

- User-mode Signal APIs, *Cont'd*
  - ***kill()*** : sends signal to a process.
    int kill(pid_t pid, int sig);

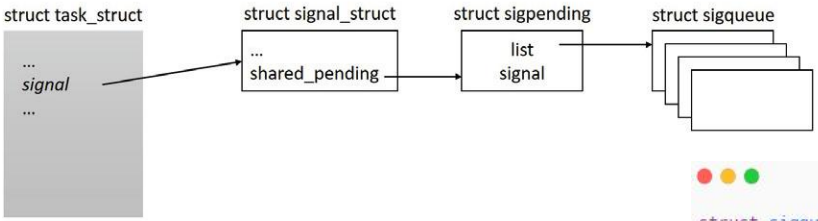
  - ***sigqueue()***
    int sigqueue(pid_t pid, int sig, const union sigval value);
    sends the signal specified in ***sig*** to the process whose PID is given in ***pid*** and the ***value*** argument is used to specify an accompanying item of data to be sent with the signal.

  - Waiting for signals,
    int sigsuspend(const sigset_t *mask);
    temporarily replaces the signal mask of the calling thread with the ***mask*** and then suspends the thread until delivery of a signal.

    int sigwaitinfo(const sigset_t *restrict set, siginfo_t *restrict info);
    suspends execution of the calling thread until one of the signals in ***set*** is pending and return with info about the pending signal.

Signals

- Signal kernel structures
  - Each LWP (thread) maintains its own pending, and blocked signal queues.
  - ***signal*** : points to the task structure ***signal_struct*** which is the signal descriptor.
    This structure is shared by all LWPs of a thread group and maintains elements such as a ***shared_pending*** signal queue (for signals queued to a thread group), which is common to all threads in a process group.




```c
struct sigqueue {
    struct list_head list;
    int flags;
    kernel_siginfo_t info;
    struct ucounts *ucounts;
};
```

```c
struct task_struct {
    ...
    /* Signal handlers: */
    struct signal_struct        *signal;
    struct sighand_struct __rcu     *sighand;
    sigset_t            blocked;
    sigset_t            real_blocked;
    /* Restored if set_restore_sigmask() was used: */
    sigset_t            saved_sigmask;
    struct sigpending       pending;
    unsigned long           sas_ss_sp;
    size_t              sas_ss_size;
    unsigned int            sas_ss_flags;
    ...
}
```

/include/linux/sched/signal.h

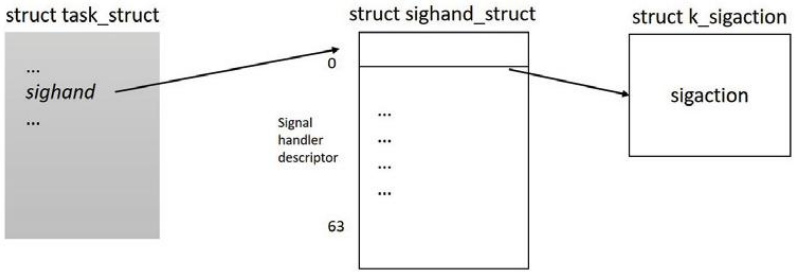
```c
struct signal_struct {
    ...
    /* shared signal handling: */
    struct sigpending   shared_pending;
    ...
}

struct sigpending {
    struct list_head list;
    sigset_t signal;
};
```

Signals

- Signal kernel structures, *Cont'd*
  - ***sighand*** : points to ***sighand_struct*** which is the signal handler descriptor shared by all processes in a thread group.

    sighand_struct : has an array of actions ***action[]*** which contains ***sigaction*** that describes the current handlers of each signal.




```c
struct task_struct {
    ...
    /* Signal handlers: */
    struct signal_struct        *signal;
    struct sighand_struct __rcu     *sighand;
    sigset_t            blocked;
    sigset_t            real_blocked;
    /* Restored if set_restore_sigmask() was used: */
    sigset_t            saved_sigmask;
    struct sigpending       pending;
    unsigned long           sas_ss_sp;
    size_t              sas_ss_size;
    unsigned int            sas_ss_flags;
    ...
}
```

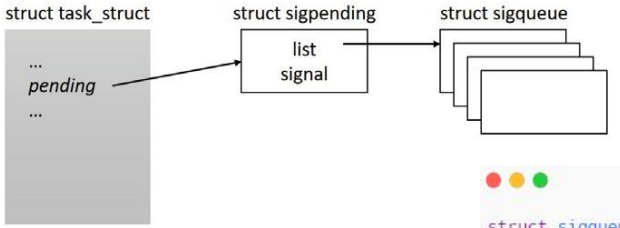
/include/linux/sched/signal.h

```c
struct sighand_struct {
    spinlock_t      siglock;
    refcount_t      count;
    wait_queue_head_t   signalfd_wqh;
    struct k_sigaction  action[_NSIG];
};

struct k_sigaction {
    struct sigaction sa;
#ifdef __ARCH_HAS_KA_RESTORER
    __sigrestore_t ka_restorer;
#endif
};
```

Signals

- Signal kernel structures*, Cont'd*
  - ***blocked, real_blocked*** : are bit masks of blocked signals. Each LWP in a thread group thus has its own blocked signal mask.
  - ***pending*** : queue private pending signals; all signals queued to a normal process and a specific LWP in a thread group are queued into list ***sigpending.***




```c
struct sigqueue {
    struct list_head list;
    int flags;
    kernel_siginfo_t info;
    struct ucounts *ucounts;
};
```

```c
struct task_struct {
    ...
    /* Signal handlers: */
    struct signal_struct        *signal;
    struct sighand_struct __rcu     *sighand;
    sigset_t            blocked;
    sigset_t            real_blocked;
    /* Restored if set_restore_sigmask() was used: */
    sigset_t            saved_sigmask;
    struct sigpending       pending;
    unsigned long           sas_ss_sp;
    size_t              sas_ss_size;
    unsigned int            sas_ss_flags;
    ...
}
```

/include/linux/signal_types.h

```c
struct sigpending {
    struct list_head list;
    sigset_t signal;
};
```

Signals

- Signal generation and delivery
  - A signal is said to be **generated** when its occurrence is enqueued, to list of pending signals in the task structure of the receiver process or processes.
  - A signal **delivery** is equal to initialization of the corresponding handler.
  - Generally signal is generated upon request from a user-mode process, or any kernel code.
  - Signal-generation calls from kernel
    send_sig() : Generates a specified signal on a process.
    send_sig_info() : Extends ***send_sig()*** with additional ***siginfo_t*** instances.

    force_sig() : Used to generate priority non-maskable signals which cannot be ignored or blocked
    force_sig_info() : Extends ***force_sig()*** with additional ***siginfo_t*** instances.

/kernel/signal.c

```
int send_sig(int sig, struct task_struct *p, int priv)
{

}

int send_sig_info(int sig, struct kernel_siginfo *info, struct
task_struct *p)
{

}

void force_sig(int sig)
{

}

int force_sig_info(struct kernel_siginfo *info)
{

}
```

Signals

- Signal generation and delivery, *Cont'd*
  - Signal-generation calls from kernel, *Cont'd*

    kill_pid() : Generates the specified signal to a thread group identified by a PID.

    kill_pid_info() : Extends ***kill_pid()*** with additional ***siginfo_t*** instances.

  - Signal delivery
    Priority signals ***SIGSTOP*** and ***SIGKILL*** are delivered even if the receiver is not on CPU by waking up the process; however, for the rest of the signals, delivery is deferred until the process is ready to receive signals.

/kernel/signal.c

```
int kill_pgrp(struct pid *pid, int sig, int priv)
{

}

int kill_pid(struct pid *pid, int sig, int priv)
{

}

int kill_pid_info(int sig, struct kernel_siginfo *info, struct
pid *pid)
{

}
```

Signals

- Signal generation and delivery, *Cont'd*
  - Signal delivery, *Cont'd*

    Delivery Steps

    1) The kernel checks for nonblocked pending signals of a process on return from interrupt and system calls before allowing a process to resume user-mode execution.

    2) The kernel function ***do_signal(struct pt_regs *regs)*** is invoked with the user-mode registers state of the process ***regs*** which stored in the process kernel stack, to initiate delivery of the pending signal before resuming the user-mode context of the process.

/arch/arm64/kernel/signal.c

```
static void do_signal(struct pt_regs *regs)
{
    unsigned long continue_addr = 0, restart_addr = 0;
    int retval = 0;
    struct ksignal ksig;
    bool syscall = in_syscall(regs);

    /*
     * If we were from a system call, check for system call restarting...
     */
    if (syscall) {
        ...
    }

    /*
     * Get the signal to deliver. When running under ptrace, at this point
     * the debugger may change all of our registers.
     */
    if (get_signal(&ksig)) {
        /*
         * Depending on the signal settings, we may need to revert the
         * decision to restart the system call, but skip this if a
         * debugger has chosen to restart at a different PC.
         */
        if (regs->pc == restart_addr &&
            (retval == -ERESTARTNOHAND ||
             retval == -ERESTART_RESTARTBLOCK ||
             (retval == -ERESTARTSYS &&
              !(ksig.ka.sa.sa_flags & SA_RESTART)))) {
            syscall_set_return_value(current, regs, -EINTR, 0);
            regs->pc = continue_addr;
        }

        handle_signal(&ksig, regs);
        return;
    }
    ...
}
```

Signals

- Signal generation and delivery, *Cont'd*
  - Executing user-mode handlers

    1) ***do_signal()*** invokes the ***handle_signal()*** routine for delivery of pending signals whose handler is set to user handler.

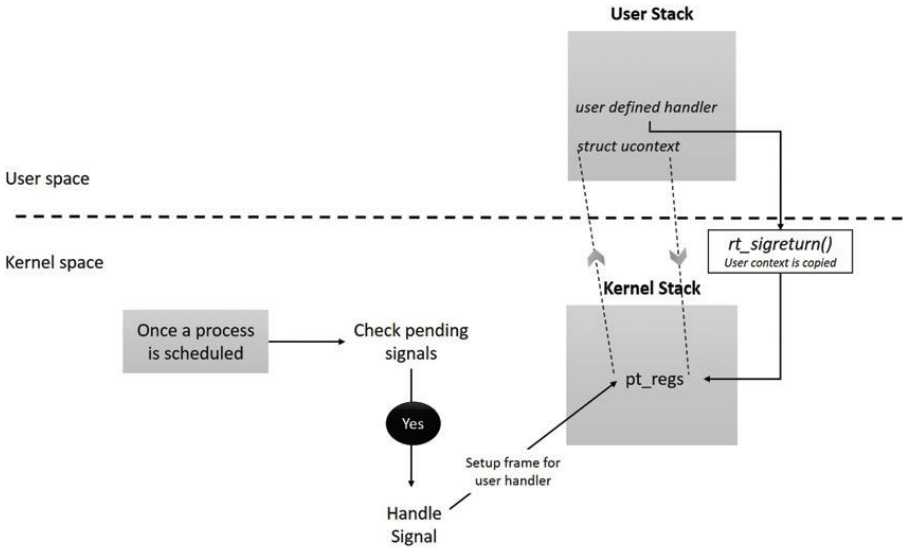
    2) But, the user-mode signal handler resides in the process code segment and requires access to the user-mode stack of the process; therefore, the kernel needs to **switch** to the user-mode stack for executing the signal handler.

    3) Switch back to the kernel stack to restore the user context for normal user-mode execution.
    But, now the kernel no longer contains the user context ***ptr_regs *reg***

    So, ***handle_signal()*** moves as well the user-mode hardware context ***ptr_regs *reg*** in the kernel stack into the user-mode stack and sets up the handler frame to invoke the ***setup_rt_frame*** that copies the hardware context back into the kernel stack.

/arch/arm64/kernel/signal.c

```
static void handle_signal(struct ksignal *ksig, struct pt_regs *regs)
{
    ...
    ret = setup_rt_frame(usig, ksig, oldset, regs);
    ...
}
```



Signals

- Signal generation and delivery, *Cont'd*
  - Executing user-mode handlers, *Cont'd*
    5) ***setup_rt_frame()*** *invokes* ***setup_return()*** which copies the hardware context back into the kernel stack and restores the usermode context for resuming normal execution of the current process.

/arch/arm64/kernel/signal.c

```
static int setup_rt_frame(int usig, struct ksignal *ksig, sigset_t *set,
              struct pt_regs *regs)
{
    ...
    if (err == 0) {
        setup_return(regs, &ksig->ka, &user, usig);
        if (ksig->ka.sa.sa_flags & SA_SIGINFO) {
            err |= copy_siginfo_to_user(&frame->info, &ksig->info);
            regs->regs[1] = (unsigned long)&frame->info;
            regs->regs[2] = (unsigned long)&frame->uc;
        }
    }

    return err;
}
```