The Linux Kernel: Signals & Interrupts
Signals

- Introduced in UNIX systems to simplify IPC.
- Used by the kernel to notify processes of system events.
- A signal is a short message sent to a process, or group of processes, containing the number identifying the signal.
  - No data is delivered with traditional signals.
  - POSIX.4 defines i/f for queueing & ordering RT signals w/ arguments.
Example Signals

- Linux supports 31 non-real-time signals.
- POSIX standard defines a range of values for RT signals:
  - `SIGRTMIN 32 ... SIGRTMAX (_NSIG-1)` in `<asm-*/signal.h>`

<table>
<thead>
<tr>
<th>#</th>
<th>Signal Name</th>
<th>Default Action</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIGHUP</td>
<td>Abort</td>
<td>Hangup terminal or process</td>
</tr>
<tr>
<td>2</td>
<td>SIGINT</td>
<td>Abort</td>
<td>Keyboard interrupt (usually Ctrl-C)</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>SIGKILL</td>
<td>Abort</td>
<td>Forced process termination</td>
</tr>
<tr>
<td>10</td>
<td>SIGUSR1</td>
<td>Abort</td>
<td>Process specific</td>
</tr>
<tr>
<td>11</td>
<td>SIGSEGV</td>
<td>Dump</td>
<td>Invalid memory reference</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Signal Transmission

- **Signal sending:**
  - Kernel updates descriptor of destination process.

- **Signal receiving:**
  - Kernel forces target process to “handle” signal.
  - *Pending signals* are sent but not yet received.
  - Up to one pending signal per type for each process, except for POSIX.4 signals.
  - Subsequent signals are discarded.
  - Signals can be blocked, i.e., prevented from being received.
Signal-Related Data Structures

- `sigset_t` stores array of signals sent to a process.
- The process descriptor (structure `task_struct` in `<linux/sched.h>`) has several fields for tracking sent, blocked and pending signals.

```c
struct sigaction {
    void (*sa_handler)();  /* handler address, or SIG_IGN, or SIG_DFL */
    sigset_t sa_mask;      /* blocked signal list */
    int sa_flags;          /* options e.g., SA_RESTART */
}
```
Sending Signals

- A signal is sent due to occurrence of corresponding event (see `kernel/signal.c`).
  - e.g., `send_sig_info(int sig, struct siginfo *info, struct task_struct *t);
    - `sig` is signal number.
    - `info` is either:
      - address of RT signal structure.
      - 0, if user mode process is signal sender.
      - 1, if kernel is signal sender.
  - e.g., `kill_proc_info(int sig, struct siginfo *info, pid_t pid);`
Receiving Signals

- Before process $p$ resumes execution in user mode, kernel checks for pending non-blocked signals for $p$.
  - Done in `entry.S` by call to `ret_from_intr()`, which is invoked after handling an interrupt or exception.
- `do_signal()` repeatedly invokes `dequeue_signal()` until no more non-blocked pending signals are left.
- If the signal is not ignored, or the default action is not performed, the signal must be *caught*. 
Catching Signals

- `handle_signal()` is invoked by `do_signal()` to execute the process’s registered signal handler.
- Signal handlers reside (& run) in user mode code segments.
  - `handle_signal()` runs in kernel mode.
  - Process first executes signal handler in user mode before resuming “normal” execution.
- Note: Signal handlers can issue system calls.
  - Makes signal mechanism complicated.
  - Where do we stack state info while crossing kernel-user boundary?
Re-execution of System Calls

- “Slow” syscalls e.g. blocking read/write, put processes into waiting state:
  - TASK_(UN) INTERRUPTIBLE.
  - A task in state TASK_INTERRUPTIBLE will be changed to the TASK_RUNNING state by a signal.
  - TASK_RUNNING means a process can be scheduled.
    - If executed, its signal handler will be run before completion of “slow” syscall.
    - The syscall does not complete by default.
    - If SA_RESTART flag set, syscall is restarted after signal handler finishes.
Real-Time Signals

- Real-Time signals are queued as a list of `signal_queue` elements:

  ```
  struct signal_queue {
    struct signal_queue *next;
    siginfo_t info; /* See asm-*//siginfo.h */
  }
  ```

- A process’s descriptor has a `sigqueue` field that points to the first member of the RT signal queue.
- `send_sig_info()` enqueues RT signals in a `signal_queue`.
- `dequeue_signal()` removes the RT signal.
RT Signal Parameters

- `siginfo_t` contains a member for RT signals.
- The argument to RT signals is a `sigval_t` type:

  ```c
  typedef union sigval {
    int sigval_int;
    void *sival_ptr;
  } sigval_t;
  ```

- Extensions?
  - Explicit scheduling of signals and corresponding processes.
Signal Handling System Calls

- int sigaction(int sig, const struct sigaction *act, struct sigaction *oact);
  - Replaces the old signal() function.
  - Used to bind a handler to a signal.
  - For RT signals, the handler’s prototype is of form:
    - void (*sa_sigaction)(int, siginfo_t *, void *);
- See Steven’s “Advanced Programming in the UNIX Environment” for more…
Interrupts

- Interrupts are events that alter sequence of instructions executed by a processor.
  
  **Maskable interrupts:**
  - Sent to **INTR** pin of x86 processor. Disabled by clearing **IF** flag of eflags register.

  **Non-maskable interrupts:**
  - Sent to **NMI** pin of x86 processor. Not disabled by clearing **IF** flag.

- **Exceptions:**
  - Can be caused by faults, traps, programmed exceptions (e.g., syscalls) & hardware failures.
Interrupt & Exception Vectors

- 256 8-bit vectors on x86 (0..255):
  - Identify each interrupt or exception.
- Vectors:
  - 0..31 for exceptions & non-maskable interrupts.
  - 32..47 for interrupts caused by IRQs.
  - 48..255 for “software interrupts”.
    - Linux uses vector 128 (0x80) for system calls.
IRQs & Interrupts

- Hardware device controllers that issue interrupt requests, do so on an IRQ (Interrupt ReQuest) line.
- IRQ lines connect to input pins of interrupt controller (e.g., 8259A PIC).
- Interrupt controller repeatedly:
  - Monitors IRQ lines for raised signals.
  - Converts signal to vector & stores it in an I/O port for CPU to access via data bus.
  - Sends signal to INTR pin of CPU.
  - Clears INTR line upon receipt of ack from CPU on designated I/O port.
Example Exceptions

<table>
<thead>
<tr>
<th>#</th>
<th>Exception</th>
<th>Exception Handler</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Divide Error</td>
<td>divide_error()</td>
<td>SIGFPE</td>
</tr>
<tr>
<td>1</td>
<td>Debug</td>
<td>debug()</td>
<td>SIGTRAP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Invalid Opcode</td>
<td>invalip_op()</td>
<td>SIGILL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Page Fault</td>
<td>page_fault()</td>
<td>SIGSEGV</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Interrupt Descriptor Table

- A system *Interrupt Descriptor Table* (IDT) maps each vector to an interrupt or exception handler.
  - IDT has up to 256 8-byte *descriptor entries*.
  - `idtr` register on x86 holds base address of IDT.
- Linux uses two types of descriptors:
  - *Interrupt gates* & *trap gates*.
    - Gate descriptors identify address of interrupt / exception handlers
    - Interrupt gates clear IF flag, trap gates don’t.
Interrupt Handling

- CPU checks for interrupts after executing each instruction.
- If interrupt occurred, control unit:
  - Determines vector \( i \), corresponding to interrupt.
  - Reads \( ith \) entry of IDT referenced by \( idtr \).
    - IDT entry contains a segment selector, identifying a segment descriptor in the global descriptor table (GDT), that identifies a memory segment holding handler fn.
  - Checks interrupt was issued by authorized source.
Interrupt Handling …continued…

- Control Unit then:
  - Checks for a change in privilege level.
    - If necessary, switches to new stack by:
      - Loading \( \text{ss} \) & \( \text{esp} \) regs with values found in the *task state segment* (TSS) of current process.
      - Saving old \( \text{ss} \) & \( \text{esp} \) values.
  - Saves state on stack including \textit{eflags}, \textit{cs} & \textit{eip}.
  - Loads \( \text{cs} \) & \textit{eip} w/ segment selector & offset fields of gate descriptor in \textit{ith} entry of \textit{IDT}.
- Interrupt handler is then executed!
Protection Issues

- A *general protection exception* occurs if:
  - Interrupt handler has lower privilege level than a program causing interrupt.
  - Applications attempt to access interrupt or trap gates.
  - *What would it take to vector interrupts to user level?*

- Programs execute with a current privilege level (CPL).
- e.g., If gate descriptor privilege level (DPL) is lower than CPL, a general protection fault occurs.
Gates, Gates but NOT *Bill Gates*!

- Linux uses the following gate descriptors:
  - **Interrupt gate:**
    - DPL=0, so cannot be accessed by user mode progs.
  - **System gate:**
    - DPL=3, so can be accessed by user mode progs.
    - e.g., vector 128 accessed via syscall triggered by int 0x80.
  - **Trap gate:**
    - DPL=0. Trap gates are used for activating exception handlers.
Initializing IDT

- Linux uses the following functions:
  - `set_intr_gate(n, addr);`
  - `set_trap_gate(n, addr);`
  - `set_system_gate(n, addr);`

- Insert gate descriptor into \textit{nth} entry of IDT.
- \texttt{addr} identifies offset in kernel’s code segment, which is base address of interrupt handler.
- DPL value depends on which fn (above) is called.

- e.g.,
  
  ```c
  set_system_gate(0x80, &system_call);
  ```