

Signals and Session Management

Signals

- Mechanism to notify processes of system events
- Primitives for communication and synchronization between user processes

Signal generation and handling

- Allow an action to be performed when an event occurs
 - Events are defined by integers mapped to symbolic constants
 - * Symbolic constants help preserve the portability of code
 - Events can be asynchronous or synchronous
- Two phases of signaling process
 1. Signal generation
 - Occurrence of event that requires notification to a process
 2. Signal delivery
 - Signal is recognized by the process and appropriate action is performed
 - Signal is *pending* between generation and delivery
- Signal handling
 - Default action for signal performed by kernel when the process does not specify alternative
 - Five possible default actions
 1. **abort**
 - * Terminates the process after dumping core
 - * Process's address space and register context is written to a file called **core** in the process's current working directory
 2. **exit**
 - * Terminate the process without generating core dump
 3. **ignore**
 - * Ignore all signals
 4. **stop**
 - * Suspend the process
 5. **continue**
 - * Resume a suspended process
 - Process can override the default action and specify an alternative signal handler method
 - A process may temporarily block a signal
 - * A blocked signal is not delivered until it is unblocked
 - * User cannot ignore, block, or specify an alternative handler for **SIGKILL** and **SIGSTOP**
 - Any signal handling action, including process termination, is performed by the receiving process itself
 - * Action can be taken only when the process is scheduled to run
 - * On a busy system, a low priority process may take a while to respond to a signal
 - * Problem may be compounded if the process is swapped out, suspended, or blocked
 - Process becomes aware of signal when kernel calls **issig()** on its behalf

- Kernel calls `issig()`
 - * Before returning to user mode from system call or interrupt
 - * Just before blocking on an interruptible event
 - * Immediately after waking up from an interruptible event
 - If `issig()` returns true, kernel calls `psig()` to dispatch the signal who
 - * terminates the process, generating core file if needed
 - * or calls `sendsig()` to invoke user-defined signal handler
 - `sendsig()`
 - * returns the process to user mode
 - * transfers control to signal handler
 - * arranges for the process to resume the interrupted code after signal handler completes
 - If signal comes in the middle of system call, system call aborts and returns `EINTR`
- Signal generation
 - Major signal sources because of which kernel generates signals are:
 - * Exception – Attempt to execute an illegal instruction
 - * Other processes – Signal from one process to another through `kill` or `sigsend` system calls
 - * Terminal interrupts – Signals for foreground processes, such as `^C`, `^\`, and `^Z`
 - * Job control – Signals for the background processes attached to a terminal
 - * Quotas – Signal sent by kernel when a process exceeds its limits for resources (check `limit(1)` man page)
 - * Notifications – Request by a process for being informed of events such as device being ready
 - * Alarms – Set for a certain time so that kernel informs the process via a signal upon expiry of that time period
 - `ITIMER_REAL` measures the real clock time and generates `SIGALRM`
 - `ITIMER_VIRTUAL` measures the virtual clock time (when the process runs in user mode) and generates `SIGVTALRM`
 - `ITIMER_PROF` measures the total time used by the process in user and kernel modes, and generates `SIGPROF`
 - Typical scenarios
 - Terminal interrupt
 - Exceptions
 - Sleep and signals
 - Should the sleeping process be awakened to receive the signal?
 - Disk I/O vs. keyboard character wait
 - Uninterruptible sleep
 - * Process sleeps for short term event like disk I/O
 - * Cannot be disturbed by the signal
 - * Signal generated for the process is marked as pending without any further action
 - * Process notices signal only when it is about to return to user mode or block on an interruptible event
 - Interruptible sleep
 - * Process waiting for an event that may not occur for a long time
 - * Wake up the process if there is a signal for it
 - Process about to block on interruptible event checks for signals just before blocking

- * If a signal is found, it is handled and system call is aborted
- * A signal after blocking the process will make the kernel to wake up the process
- * The awakened process will first call `issig()` to check for signal
- * `issig()` is always followed by `psig()` to check for pending signal

Unreliable signals

- Original implementation of signals (prior to SVR2) is unreliable
 - Problem with signal delivery
 - Signal handlers are not persistent and do not mask recurring instances of same signal
 - After signal occurrence, kernel resets the signal action to default
 - Users must reinstall signal handlers after each signal occurrence leading to race condition
 - * Suppose user hits **CTRL-C** twice in quick succession
 - * First **CTRL-C** resets the signal handler action to default and invokes the handler
 - * Second **CTRL-C** may not be caught if the handler is not installed right away
 - * This is why these signals are called *unreliable*
 - Performance problem with sleeping processes
 - * All information regarding signal handling is stored in `u_signal[]` in `u` area, with one entry for each signal type
 - * The entry contains the address of user-defined handler, or `SIG_DFL` to specify the default action, or `SIG_IGN` to ignore the signal
 - * Kernel passes the signal to process to deal with because it cannot read the `u` area of a process that is not current process
 - If the process is sleeping, kernel wakes it up
 - If the process is to ignore the signal, it simply does so and goes back to sleep
- SVR2 lacks a facility to block a signal temporarily
- SVR2 also lacks job control

Reliable signals

- Primary features
 - Persistent handlers
 - * Signal handlers are not reset to default after handling a signal
 - Masking
 - * A signal can be masked/blocked temporarily
 - * Kernel will remember that the signal is blocked and not immediately post it to the process
 - * Signal will be posted when the process unblocks
 - * This can be used to protect critical regions of the code from being interrupted by signals
 - Sleeping processes
 - * Signal handling information can be kept in `proc` area instead of `u` area to make it visible to kernel
 - Unblock and wait
 - * Process is blocked by `pause(2)` until a signal arrives

- * Another function – `sigpause(2)` automatically unmask a signal and blocks the process until the signal is received
- SVR3 implementation
 - `sigpause(2)` system call
 - * Let a process declare a handler for `SIGQUIT` signal and set a global flag when the signal is caught
 - * Process waits for the flag to be set (critical section)
 - * If signal arrives after check but before wait, it will be missed and process will wait forever
 - * Process should mask `SIGQUIT` while testing the flag
 - * If it enters wait with masked signal, signal can never be delivered
 - * `sigpause(2)` unmask the signal and blocks the process atomically
 - SVR3 lacks support for job control and facilities for automatic restart of system calls
- BSD signal management
 - Most system calls take a 32-bit signal mask argument, one bit per signal
 - * A single call can operate on multiple signals
 - * `sigsetmask(3B)` specifies the set of signals to be blocked
 - * One or more signals can be added to the set using `sigblock(3B)`
 - * In BSD, `sigpause(2)` automatically installs a new mask of blocked signals and puts the process to sleep until a signal arrives
 - * `sigvec(3B)` installs a handler for one signal, and can specify a mask to be associated with it
 - * When a signal is generated, kernel will install a new mask of blocked signal that contains current mask, mask specified by `sigvec(3B)` and current signal
 - Handler always runs with current signal blocked so that a second instance of the signal will not be delivered until the handler completes
 - When the handler returns, blocked signals mask is restored to its previous value
 - Signals are handled on a separate stack
 - * Processes may manage their own stack so that the process stack is also shared for signals
 - * Stack overflow itself may cause a `SIGSEGV` exception
 - * Running signal handlers on separate stack may resolve this problem
 - * C library function `sigstack(3C)` allows the calling process to indicate to the system an area of its address space to be used for processing signals
 - * User should make sure that the stack is large enough as the kernel does not know stack bound
 - Additional signals
 - * Required for tasks like job control
 - * User can run several processes, with at most one being in the foreground
 - * Different shells use signals to move jobs between foreground and background
 - Automatic restart of system calls
 - * Allowed for slow calls that may be aborted by signals
 - * Exemplified by `read(2)` and `write(2)`
 - * These calls restart after the handler returns instead of being aborted with `EINTR`
 - * `siginterrupt(3B)` allows signals to interrupt functions, and to change the function restart behavior

Signals in SVR4

- System calls provide a superset of SVR3 and BSD signal functionality

- Compatibility interface with older releases is provided through library functions (check out the man sections of calls in previous sections)
- Directly correspond to the POSIX.1 functions in name, calling syntax, and semantics

Signals implementation

- Kernel must maintain some state in both the **u** area and the **proc** structure for efficiency
 - **u** area contains information required to properly invoke signal handlers, using the following fields
 - * **u_signal[]** – Vector of signal handlers for each signal
 - * **u_sigmask[]** – Signal masks for each handler
 - * **u_signalstack** – Pointer to alternate signal stack
 - * **u_sigonstack** – Mask of signals to handle on alternate stack
 - * **u_oldsig** – Set of handlers to exhibit unreliable signals
 - **proc** structure contains fields related to generation and posting of signals, with the following fields
 - * **p_cursig** – Current signal being handled
 - * **p_sig** – Pending signals mask
 - * **p_hold** – Blocked signals mask
 - * **p_ignore** – Ignored signals mask
- Signal generation
 - Kernel checks the **proc** structure of the receiving process
 - Is signal ignored? If yes, kernel just returns
 - If not, kernel adds the signal to the set of pending signals in **p_cursig**
 - * Multiple instances of same signal cannot be recorded
 - Process will only know that at least one instance of the signal was pending
 - Process in interruptible sleep is awakened to deliver the signal if the signal is not blocked
 - Job control signals (**SIGSTOP**, **SIGSUSP**, and **SIGCONT**) directly suspend or resume the process instead of being posted
- Delivery and handling
 - Process checks for signal using **issig()**
 - * When about to return from kernel mode after system call or interrupt
 - * At the beginning or end of interruptible sleep
 - **issig()** looks for set bits in **p_cursig**, the current signal being handled
 - * If any bit is set, **issig()** checks **p_hold** (blocked signal mask) to see if the signal is currently blocked
 - * If signal is not blocked, **issig()** stores the signal number in **p_sig** (pending signal mask) and returns **true**
 - If a signal is pending, kernel calls **psig()** to handle it
 - * **psig()** checks information in **u** area for the signal
 - * If there is no handler, **psig()** takes the default action, possibly process termination
 - * If there is a handler, **psig()** adds current signal to **p_hold** (blocked signals mask), as well as any signal specified in the **u_sigmask[]** vector (signals corresponding to the handler)
 - * Current signal is not added if **SA_NODEFER** flag is specified for the handler
 - * If **SA_RESETHAND** flag is specified, action in the **u_signal[]** vector is set to **SIG_DFL**
 - Finally, **psig()** calls **sendsig()**

- * **send_sig()** arranges for process to return to user mode and pass control to handler
- * When handler completes, process resumes code being executed prior to receiving the signal
- * If alternate stack is to be used, **send_sig()** invokes the handler on that stack

Exceptions