Lecture Notes - Signals

Debadatta Mishra
Indian Institute of Technology Kanpur

Signals

- In many scenarios, either the OS or some other processes may want to send some notifications to a particular user process. For example, if a process is executing an infinite loop because of programming errors, the user may want to kill the process (from a shell). Further, the OS may want to notify the process if it performs a division-by-zero so that the process can perform cleanup activities before exiting. There are other situations where a process may require a notification to realize certain functionalities. For example, a parent process may want a notification from the OS when one of its child processes terminates.
- UNIX OSs provide the signal system call API to enable user processes register notification handlers to different events. The signal system call allows a user process to register handlers for different events (signals). The signature of signal system call is as follows,

  signal (signum, sighandler) where **signum** is an integer specifying the signal (event) for which the user process wants notification from the OS. Some examples of signal numbers are, SIGINT, SIGHUP, SIGALRM, SIGCHLD etc. **sighandler** is the function pointer of the handler which is invoked when the signal event occurs. For example, if **signal (SIGINT, sigint_handler)** is invoked from a process, the **sigint_handler** function is executed every time CTRL+C is pressed on the console where the process is executing. The value of **sighandler** can be SIG_DFL to specify default handler (inside OS) or SIG_IGN to ignore the signal. Note that, for some signals (like SIGKILL, SIGQUIT), a user process is not allowed to ignore or register a custom handler. For more details, refer to the man pages (man signal).
- A signal can be sent from a process to another process using the **kill** system call. **kill(pid, SIGSEGV)** sends the SEGV (segfault) signal to the process with PID=pid. If pid is zero, the signal will be delivered to all processes in the calling process group. Note that, all child processes belong to the same process group of the parent unless the child executes setpgrp( ) system call to set its own process group.
- The OS maintains a per-process vector of handlers corresponding to all signals in the process control block (PCB). The value of handler can be one of the following: Default handlers, no handler (ignore) or custom handler (registered by the process using **signal( )** system call). Further, the OS also maintains a pending signal bit vector where each bit corresponds to a signal which is either set by the OS to indicate an event like SEGFAULT or by another process through the **kill( )** system call.
- Signals can be delivered in a synchronous manner when the event occurs because of any misbehavior of the executing process. For example, if a process causes an exception (by performing division by zero), the signal may be delivered when returning to the user space after the exception handling in the OS. In many other

cases, if the process is executing in user mode when the signal event occurs, the signal has to be delivered in a deferred (asynchronous) manner. For example, if process A wants to send a signal to process B which is executing in user mode, the signal pending bit is set to one (in the PCB of B) and the signal is sent to B when the OS gains the execution control.

- The OS can invoke the signal handler of a process when it is returning from an exception handler, system call handler or an interrupt handler. Therefore, the maximum time the signal remains undelivered depends on two factors---(i) the time till the OS gains control from the process and, (ii) the time till the process gets scheduled again.
- In case of a custom signal handlers, the OS returns to the user process resuming execution of the handler function after which the control comes back to the point where the process was executing before. To realize this behavior, the OS must mimic a function call in the user space. This requires careful manipulation of the user space stack and the instruction pointers by the OS. One possible design can be to create a new stack frame with return address set to the original user execution point (value of RIP before the entry into OS through a syscall, exception or an interrupt). On X86 systems, this is possible as the user process execution details (user RSP, RIP etc.) are pushed to the OS stack and the OS can access and appropriately augment them.
- An important design consideration for the OS is to design signal delivery mechanisms in presence of nested signals. More specifically, if a signal event occurs while one instance of the same signal is being handled, what should the OS do? If the signals are delivered during signal handling, the OS needs to carefully manage the user stack and further, the user stack may overflow in case of multiple nested invocation of signal handlers. Another alternative can be to disable signals during the execution of signal handlers and the user process re-registers the signal after handling the signal. There are multiple issues with this approach---lost signals, burden on the programmer to re-enable etc.
- Modern days OSs like Linux, insert system calls in the execution flow in a user transparent manner where the signal gets re-enabled through a special system call like **sigreturn( )** when the signal handler finishes**.** This can be done using techniques similar to signal handler call mimicking through user stack manipulations or creating a temporary stack (a.k.a. Signal stack)
- Signal handlers are inherited by the child processes. However, the pending signals (if any) bit vector is all cleared for the child and are not delivered to the child. This makes sense as the pending signals are meant for the parent because they are set before the child creation.