Reliable and unreliable signals in unix pdf

Conference Notes: Chapter huangj/cs360/360/notes/Signals/lecture.html 10 of the book provides a very complete description of the signs. It will be a better read after reading these conference notes, but you will benefit from reading it though. Signals are a complex control flow operation. A signal is a program interruption of some kind. To fully and correctly understand signals, one should ALWAYS REMEMBER that signals are classic examples of asynchronous events. For example, when you hit CNTL-C, it sends the SIGINT signal to your program. When you hit the CNTL-, it sends the SIGQUIT signal to your program. When a segmentation violation is generated, it sends the SIGSEGV signal to the program. In any case, the process does not know a priori when and if the signal will occur. Before discussing the technical details, let's get acquainted with some names that occur in each documentation about the signal. This is Unix Version 7 (abbreviated as Version 7), SVR4 (System V Release 4), and BSD 4.x (Berkeley Software Distribution version 4.x), POSIX. 1 (Portable operating system interface, backed by IEEE). Version 7 was the version of Unix released in 1979, when Unix was still developed in an open source environment. During that time period, BSD already existed and made a great contribution to the development of Unix in general. (Bill Joy, a key member of the BSD group co-founded Sun Microsystems in 1982). In 1983, AT&T was quick to market The Unix and relabeled the system as System V. BSD still remained, but quickly began to diverge with the V system. Full Unix standardization has never occurred after the 1983 turning point. IEEE launched its efforts to develop a portable operating system and ended up with a standard, which practically covers the terrain of comman among the main flavors of Unix. What do all these have to do with the signal? Well, the first different systems have a different number of signals, and very often a slightly different set of signals as well. For example, version 7 has only 15 different signals, but SVR4 has 31 signals. ANSI C also defines a very small set of signals, which form a subset of POSIX signals. However, the ANSI C standard has made the signal so general that it is often considered useless. The bad news here is, C programs that use signals for even moderately complicated purposes is not likely to be highly portable between different systems. Second, as you'll probably read on the man pages, there are two signal models, one unreliable and one reliable. Version 7 used the unreliable (with good intentions leading to problems). SVR4, BSD4.3+ and POSIX, etc., have adopted the reliable model. Knowing this difference would make your reading pages easier. Your program has several ways to deal with signals. All signals have names, each starting with three letters: GIS. By default, there are certain actions that take place. For example, when you hit CNTL-C, the por lo general sale. Esa es la acción predeterminada para SIGINT. Cuando se pulsa CNTL- o se obtiene una infracción de segmentación, el programa vuelca el núcleo y, a continuación, se cierra. Esa es la acción predeterminada para SIGQUIT y SIGSEGV. Puede redefinir lo que sucede cuando recibe estas señales, lo que le permite escribir programas muy flexibles. Internamente, cuando se genera una señal, el sistema operativo toma el control del programa que se está ejecutando actualmente. Guarda el estado actual del programa en la pila. A continuación, llama a un controlador de interrupción para la señal específica. Por ejemplo, el controlador de interrupciones predeterminado para SIGINT hace que el programa se cierre. El controlador de interrupciones predeterminado para SIGSEGV y SIGQUIT hace que el programa voltee el núcleo y, a continuación, salga. Si el controlador de interrupciones para una señal llama a retorno, a continuación, lo que sucede es que el sistema operativo toma el control de nuevo y restaura el programa desde el estado que ha guardado en la pila. El programa se reanuda desde donde lo dejó (generalmente - hay algunas veces cuando no lo hace). Puede utilizar la función signal() para definir controladores de interrupción para señales. Como siempre, lea la página de usuario: man 3v signal. Por ejemplo, mire sh1.c: #include < signal.h > void cntl_c_handler(int dummy) ? printf(Acabas de escribir cntl-c); signal(SIGINT, cntl_c_handler); á main() á int i, j; signal(SIGINT, cntl_c_handler <); < 1000000; i++); } } What this does is set up an interrupt handler for SIGINT. Now, when the user hits CNTL-C, the operating system will save the current execution state of the program, and then execute cntl_c_handler. When cntl_c_handler returns, the operating system resumes the program from where it was interrupted. Thus, when you run sh1, each time you type CNTL-C, it will print You just typed cntl-c, and the program will continue. It will exit by itself in 10 seconds or so. The signal handler should follow the prototype of cntl_c_handler. In other words it should return a (void) (i.e. nothing), and should accept an integer argument, even if it will not use the argument. Otherwise, gcc will complain to you. Also, note that I make a signal() call in the signal handler. On some systems (e.g. Version 7), if you do not do this, then it will reinstall the default signal handler for CNTL-C once it has handled the signal. On some systems, you don't have to make the extra signal() call. Such is life in the land of multiple Unix's. You can handle each different signal with a call to signal. For example, sh1a.c defines different signal handlers for CNTL-C (which is SIGINT), and CNTL-\ (which is SIGQUIT). They print out the values of i and j when the signal is generated. Note that i and j must be global variables for this to work. This is one example when you have to use global variables. Try this out by compiling the program and then running it, and CNTL-C and CNTL-\ a bunch of times: UNIX> sh1a ^CYou just typed cntl-c. j is 2 and i is 539943 ^CYou just typed cntl-c. j is 2 and i is 919180 ^\ Acabas de escribir escribir j is 4 and i is 413031 -CYou just type cntl-c. j is 5 and i is 20458 . He just wrote cntl. j is 6 and i is 73316 . He just wrote cntl. j is 6 and i is 683034 -CYou just type cntl-c. j is 7 and i is 292244 -CYou just type cntl-c. j is 13 and i is 738661 -You just wrote cntl-. j is 14 and i is 789583 . He just wrote cntl. j is 16 and i is 42225 -You just typed cntl-. j is 16 and i is 209458 -CUsted just typed cntl-c. j is 17 and i is 260584 . He just wrote cntl. j is 19 and i is 982514 UNIX> It can also capture the segmentation violation signal. One such CS legend is that some graduate student used to put the following in their code: #include < signal.h > #include < stdio.h > void segv_handler(int mannequin) á fprintf(stderr, nfs server not responding, still trying); while(1) ; ? main() ... signal(SIGSEGV, segv_handler(); rest of the code - This is so that if he was demonstrating his code, and there was a segmentation violation (which always seems to happen when you are demonstrating code), it would look as if the network had frozen. Very clever. (That is, look and run sh1b.c. It should cause a segmentation violation, but instead it looks like the network is hanging). Something really interesting about SIGHUP SIGHUP is a signal that the control terminal (shell process) sends to all the processes it has spawned and still possesses as children. If you have written a command such as ls, or even something more advanced like ls &, when you run these processes, the control terminal shell process is always logged by the operating system as the main process. Now, something special about shell is that it is designed to generate processes and therefore it is by design it will be a responsible process. One thing in particular: When a shell process terminates when the control terminal is closed, the shell process sends SIGHUP to all its child processes. Well, if Shell finds that there are suspended jobs in the background, for example, shell will echo a warning in stderr to remind you like this. If you insist on closing the shell, the shell will close and send those signals. With what you have been taught in this conference, you can say that this by default will kill all those child processes. So this brings to most of you that most of you should have experienced before. You log in remotely to a Unix machine, start a job (for example, you are performing an image processing lab mapping that takes some time to run), and then, during run time, the terminal was closed due to a temporary loss of wireless connection. Then, when you sign in again, your work is gone. A lot of computing time was lost. Now, as an expert in system programming, you can definitely say that the culprit is SIGHUP. Is there a way to That? Well, for a moment, you can have the childish process ignore SIGHUP. But what if this program isn't your creation. Next, let's disconnect the secondary process from the shell process. You can do it disallow or by nohup. I'll show you examples. Even better, what if the directly through shell? Try standard utilities such as at or batch. Using those, you even receive an email after the job is done. Some details about the signal when calling the signal, the two input arguments include the singal number, and a pointer to the function. One should know that the second argument, the function pointer, can be FROM SIGIGN or SIGDFL constants. If SIGIGN is passed, the process ignores the signal. Note that SIGKILL cannot be ignored, nor can it be captured, which means that a programmer configures a different controller for a signal (in case a program ruins signal control, the administrator can still delete an escape process with this SIGKILL (No. 9). If SIGDFL is passed, the process switches to the default signal handler. No matter in which case, the call signal returns the address of the previous controller for a signal. Legitimate reasons for signal generation can be classified into the following categories: 1. terminal generated from user keystrokes (CNTL-D, for example) 2. hardware exception, for example, split by 0, bus error 3. kill function (a process that sends a signal to another process or group of processes owned by the same user). Note that a process can send a signal to itself by calling to raise 4. kill command (simply a command interface to the kill function) 5. Software conditions such as SIGPIPE and SIGALARM To block signal delivery Using the sigprocmask call, a programmer can specify to block a set of signals to be delivered to a process. You can think of the mask as a bit vector, in which each bit represents a binary on/off switch. However, it is not suggested to directly manipulate each bit in the mask due to portability issues. Three standard macros are provided for accessing mask bits: SIG_BLOCK, SIG_UNBLOCK, and SIG_SETMASK. Read man -s 2 sigprocmask. If a signal is generated and its action is not SIGIGN, the signal remains pending until the process unlocks it or the action is changed to SIGIGN. That's a transient period before a legitimate signal handler is called. sigpending can be used to find out which signals are blocked and pending. To stop signal generation For some applications, we may not want some signals to be generated to get started. In that case, we need to use a data type - a set of signals. The main functions related to signal sets are: sigemptyset, sigfillset, sigaddset, sigdelset, and sigismember All are located in section 3C of the man page. Alarm() Another use of signal handles is the alarm clock provided by Unix. Read the man page for alarm(). What alarm(n) does is come back, and then n seconds late, it will cause the SIGALRM signal to occur. If you have set a signal controller for it, then you can pick up the signal, and do whatever you wanted to do. For example, sh2.c is like sh1.c only prints a message after the program has run for 3 seconds. Note that alarm() is approximate: it is not exactly 3 seconds, but we will consider it close enough for the purposes of this class>. class>. just happened: j 26. i 638663 UNIX> Finally, sh3.c shows how you can have Unix send you SIGALRM every second. It is only a setting to sh2.c where you have the alarm controller call alarm to cause Unix to generate SIGALRM one second after the current one. UNIX> sh3 1 second just passed: j 8. i 823534 2 seconds just passed: j 17. i 715735 3 seconds just passed: j 26. i 610604 4 seconds just passed: j 35. i 513675 UNIX> On some systems, when you are on a signal handler for a signal, you cannot process that same signal again until the controller returns. On other systems, you can handle that same signal again. For example, look at sh4.c. Note that the alarm_handler has an infinite loop, which means it never comes back. The program runs for one second, and then SIGALRM is generated and the alarm_handler() is entered. It enters an infinite loop, and a second later, SIGALRM is generated again. Depending on your version of Unix, different things can happen. On Solaris, the signal will be handled and alarm_handler() again. In SunOS, the signal will be ignored until you return from alarm_handler(), which of course never happens. So, here's the output on Solaris (try it on kenner): UNIX>sh4 A second has passed: j . i 697646 A second has passed: j 7. i 697646 A second has passed: j 7. i 697646 A second has passed: j 7. i 697646 ... and here's the output on SunOS (try it on duncan): UNIX> sh4 A second has passed: j 7. i 584436 You can reliably generate and handle other signals, whether on a signal controller or not. For example, when you press CNTL- on sh4.c, it detects correctly whether the program or alarm_handler() is running -- give it a try. Finally, you can send any signal to a program with the kill command. Read the man page. Signal number 9 (SIGKILL) cannot be detected by your program, which means that you cannot write a signal handler for it. This is good because if you're wrong writing a signal handler, then killing -9 is the only way to kill the program. Some interesting questions to think about. 1. When an executive is called, does the new process handle signals in the same way as the previous process does? 2. What (should) happen when a signal is activated during the process, is your signal controller currently running? Run?