Signals

A signal is a software interrupt delivered to a process. The operating system uses signals to report exceptional situations to an executing program. Some signals report errors such as references to invalid memory addresses; others report asynchronous events, such as disconnection of a phone line.

POSIX defines a variety of signal types, each for a particular kind of event. Some kinds of events make it inadvisable or impossible for the program to proceed as usual, and the corresponding signals normally abort the program. Other kinds of signals that report harmless events are ignored by default.

signal handler

If you anticipate an event that causes signals, you can define a handler function and tell the operating system to run it when that particular type of signal arrives.

Finally, one process can send a signal to another related process; this allows a process to terminate another, or two processes to communicate. Such processes must be related, i.e. share the same gid.

Types of Signals

A signal reports the occurrence of an exceptional event. These are some of the events that can cause (or generate, or raise) a signal:

- The OS reports an exceptional condition that might interest the process such as input/output completion or arrival of input from the network.
- A program error such as dividing by zero or issuing an address outside the valid range.
- A user request to interrupt or terminate the program. Most environments are set up to let a user suspend the program by typing C-z, or terminate it with C-c.
- The termination of a child process.
- Expiration of a timer or alarm.
- A call to kill or raise by the same process.
- A signal from another process. Signals are a limited but useful form of interprocess communication.

If you want to do something when a signal is delivered to your process, you have to set up a signal handler.

This is the simplest handler:

```
static void catch( int sig )
{
   fprintf( stderr , "%d ... %d\n" , getpid() , sig ) ;
   exit( 0 ) ;
}
```

the static keyword is optional.

Two default signal handlers are predefined: SIG_IGN and SIG_DFL (the latter restores the default).

A signal handler must be installed in order to be used. This is done by calling function sigaction.

```
int makechild( int t )
{
   pid_t pid ;
   struct sigaction SA ;
   if( (pid = fork()) == 0 ) {
      SA.sa_handler = catch ;
      SA.sa_flags = 0 ;
      sigaction( SIGINT , &SA , NULL ) ;
      sleep( t ) ;
      fprintf( stderr  , "%d . . . %d\n" , getpid() , t ) ;
      exit( 0 ) ;
   } else
     return pid ;
}
```

Note that sigaction will fail (and return a -1) if an attempt is made to install a handler for a signal that cannot be caught (such as SIGKILL).

POSIX defines signals and signal handling as a portable standard. Various operating systems implement the standard in their own ways, so that only the basic elements are truly portable.

As an example, take SIGTERM a signal calling for a process to terminate. Some systems block it (they should not) making it useless in the simplest circumstances.

If multiple signals of the same type are delivered to your process before your signal handler has a chance to be invoked at all, the handler may only be invoked once, as if only a single signal had arrived. In effect, the signals merge into one.

This situation can arise when the signal is blocked, or in a multiprogramming environment where the system is busy running some other processes while the signals are delivered. This means, for example, that you cannot reliably use a signal handler to count signals. The only distinction you can reliably make is whether at least one signal has arrived since a given time in the past.

Interrupts cause problems

Interrupts (and signals) sneak some code in the middle of some other code. If a process is doing something when a signal comes, the signal handler's actions will be performed right then, in the middle of the work of the process.

This is sufficiently dangerous when a process is in the middle of copying a file when it receives a SIGKILL. The file will be copied only partially.

More subtle problems can be caused by signals arriving at an unexpected moment. To avoid these problems, one should write code that is atomic where it matters.

First an example, then a partial solution.

```
struct two_words { int a, b; } memory;

void handler(int signum)
{
   printf ("%d,%d\n", memory.a, memory.b);
   alarm (1);  // Note this trick
}

int main (void)
{
   static struct two_words zeros = { 0, 0 }, ones = { 1, 1 };
   signal (SIGALRM, handler);
   memory = zeros;
   alarm (1);
   while (1) {
      memory = zeros;
      memory = ones;
   }
}
```

This program fills memory with zeros, ones, zeros, ones, alternating forever; meanwhile, once per second, the alarm signal handler prints the current contents.

Clearly, this program can print a pair of zeros or a pair of ones. But that's not all it can do! On most machines, it takes several instructions to store a new value in memory, and the value is stored one word at a time.

If the signal is delivered in between these instructions, the handler might find that memory.a is zero and memory.b is one (or vice versa).

To avoid uncertainty about interrupting access to a variable, you can use an integer data type for which access is always atomic:

sig_atomic_t

Reading and writing this data type is guaranteed to happen in a single instruction, so there's no way for a handler to run "in the middle" of an access.

The type sig_atomic_t is always an int data type, but how many bits it contains will vary from machine to machine. You can also assume that pointer types are atomic; that is very convenient. Both of these are true on all POSIX systems.

However, larger types (including objects) are not atomic and no declaration will turn them into atomic entities; the only, very partial, solution is to block signals when they are not welcome.

Blocking a signal means telling the operating system to hold it and deliver it later. Generally, a program does not block signals indefinitely–it might as well ignore them by setting their actions to SIG_IGN. But it is useful to block signals briefly, to prevent them from interrupting sensitive operations. For instance:

- You can use the sigprocmask function to block signals while you modify global variables that are also modified by the handlers for these signals.
- You can set sa_mask in your sigaction call to block certain signals while a particular signal handler runs. This way, the signal handler can run without being interrupted itself by signals.

User–defined signals

```
// When a SIGUSR1 signal arrives, set this variable.
volatile sig_atomic_t usr_interrupt = 0;

void synch_signal (int sig)
{
  usr_interrupt = 1;
}

int main (void)
{
   struct sigaction usr_action;
   usr_action.sa_handler = synch_signal;
   usr_action.sa_flags = 0;
   sigaction (SIGUSR1, &usr_action, NULL);
```

continued on the next slide

```
if( fork () == 0 ) {    // Child's code
   // Perform initialization here
   ... ... ...
   // Let parent know you're ready to work.
   kill (getppid (), SIGUSR1);
   // Continue with execution then finish
   puts ("Bye, now....");
   exit (0);
} else {    // the parent's code
   // Wait for the child to send a signal.
   while (!usr_interrupt) usleep( 100 ) ;

   // Now continue execution then finish
   puts ("That's all, folks!");
   return 0;
}
```

In the previous example only the parent needed the signal handler and it would seem natural to put the invocation of the handler inside the red code.

But this would be an error because the handler must be in place when the fork() splits the process in two–we must be prepared that the child process will execute its code before the parent executes its code (and that is the way it is on most systems).

References

There are many more details that a good programmer should know about signals.

Here are some links:

- This presentation was largely based on the GNU C Manual
- Slides from Germany
- IBM manual (excellent! Make sure to follow to the next sections using the next topic link at the end of each section).
- Finally, you may try the terse opengroup's manual.
- If the given references are not sufficient, your interests must be unbounded. My recommendation is to brighten your horizons by reading more about signals.
- If everything else fails, please consider raising the Victor signal (but don't ask for a Zulu).