Signals

Each Unix system has a fixed set of signals that one process can raise to cause an interrupt in another process that responds to a particular signal, whose response the original process can catch. A signal is an OS mechanism to notify an application process that an event has occurred. A particular hardware event may cause a signal. For example, attempted division by zero raises a signal and a client pressing the Delete key also raises a signal.

Most versions of Unix include 2 signals, named SIGUSR1 and SIGUSR2, that a client can use for application programming. The file /usr/include/bits/signum.h on a Linux box contains a complete list of the signals for Linux. Use the command “man 7 signal” to get an overview of Linux signals and how they are numbered (usually by integers with #define).

A process may respond to a signal in one of three ways:

• take the default action with SIG_DFL
• ignore the signal with SIG_IGN
• catch the signal with an address of a programmer's function.

The signal() function can be used to define default signal handling behaviors (first two options).

For example:

signal(SIGALRM, SIG_IGN) /*ignore alarms from another process*/
signal(SIGALRM, SIG_DFL) /* reset to default: enable */

To define a customized response to a signal (third option) requires that the programmer must define a function whose signature appears as the second parameter of the signal function, namely,

    void ( * sighandler )( int )

The function sighandler returns void and has one int parameter which is the signal number. The signal() function responds to its signal number parameter by calling a function named sighandler. The call requires knowledge of the address of sighandler, hence the *.

    signal(SIGALRM, alarm_handler);
Lab exercise:

Write a program where the parent forks off one child, have child send `SIGUSR1` to parent, have parent send `SIGUSR2` to child. Register handlers that will print which sigs were received (for both parent and child).

Then have parent kill child by sending `SIGTERM`. Wait on status and then report child terminated.
Kernel Time

A Unix kernel keeps the current time by reading a clock device and by maintaining a kernel variable with the current time. Current time is accessible to user mode programs via system call gettimeofday().

Time is stated relative to some important starting point. In the US this is calculated by the Gregorian calendar, which is based on a time of zero to be about 2000 years ago. When you type the date() command to the shell, the command will read the kernel variable to determine the time. Unix systems have reference point set to 12am, 1/1/1970, Greenwich time. Two long int kernel variables keep track of the number of seconds and microseconds that have elapsed since then.

#include <sys/time.h>

struct timeval t;

gettimeofday(&t, NULL);

t.tv_sec /* number of seconds since Unix Epoch */
t.tv_usec /* number of microseconds since Unix Epoch */

For tv_usec to be correct at each microsecond, Linux must access the hardware clock each microsecond. Hardware includes a programmable timer set to issue an interrupt every k time units. Linux chooses k as 10 milliseconds (called a jiffy).

The kernel also uses interval timers to keep track of three different intervals of time relevant to each process.

ITIMER_REAL passage of real time
ITIMER_VIRTUAL passage of time when the process executes (CPU time)
ITIMER_PROF virtual time + kernel time on behalf of process

An interval timer is a countdown timer, periodically initialized to some prescribed value and then reflects the passage of time by counting down towards zero. It then raises a signal (SIGALRM), resets counter and counts down again.

The system call setitimer() initializes an interval timer. it_interval field defines the value that should be used to reset the timer, it_value defines the current value for the timer. Most of the time they should be the same!

#include <sys/time.h>

struct itimerval v;

v.it_interval.tv_sec = 9;
v.it_interval.tv_usec = 999999;
v.it_value.tv_sec = 9
v.it_value.tv_usec = 999999;
Lab Exercise on Kernel Timer

Use `ITIMER_REAL` to implement a personal version of `gettimeofday()`. Set `ITIMER_REAL` to raise a signal once a second. Use `signal()` to count the number of elapsed seconds. Compare your time with the return values of `gettimeofday()`.

If you have trouble getting started, copy the skeleton code `kt_frame.c` from `~dxu/handouts/cs355` and go from there.